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

CapeVM: 用於資源受限的物聯網裝置之快速安全虛擬機

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

雷理生

Niels Reijers

指導教授：施吉昇教授

Advisor: Professor Chi-Sheng Shih

中華民國 107 年 4 月

April, 2018

國立臺灣大學博士學位論文
口試委員會審定書

CapeVM: 用於資源受限的物聯網裝置之快速安全虛擬機

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

本論文係雷理生君 (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 4 月 17 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 2.0x 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

Chapter 1

Introduction

Production of integrated circuits has advanced at a consistent and rapid pace for over half a century, doubling the transistors density every one to two years in accordance with Moore's law. Only in recent years has the cadence slowed as we approach fundamental physical limits.

While Moore's law is most commonly associated with improvements in performance, it also allowed us to reduce the size of computers: doing the same thing in ever smaller packages. This trend has not been as smooth as the continuous improvements in performance. While any improvement in performance is a direct advantage, a small reduction in size usually is not. However, it enables revolutions at certain thresholds: moving from room-sized computers to home computers and PCs in every home, scaling them down further to portable/laptop computers, and eventually handhelds and smart phones. Once established, each of these areas then benefitted from Moore's law to improve their capabilities, but the truly disruptive moments are when miniaturisation allowed whole new applications areas to emerge.

The term 'ubiquitous computing' was coined by Mark Weiser, predicting in 1991 that computing would move from a dedicated device on the desktop, to devices all around us, from alarm clocks to coffee makers [99]. Around the turn of the century, we were able able to scale down useful, working devices to the size of a few millimetres [97]. This led to the start of research into Wireless Sensor Networks (WSN): many small and inexpensive sensor nodes, often called 'motes', working together to perform continuous automated

sensing tasks.

Many promising WSN applications were proposed, ranging from military applications [5], to precision agriculture [50], habitat monitoring [62], and environmental monitoring [100, 15]. While the applications vary greatly, the hardware platforms used to build WSN applications are all quite similar, and usually very resource-constrained.

Sensor nodes are typically small and battery powered, and many applications require a lifetime measured in weeks or months rather than hours, so maintaining a very low power consumption is critical. To achieve this, the CPUs used for sensor nodes are kept very simple. While they lack most of the advanced features found in modern desktop CPUs, they typically do have several sleep modes, allowing them to reduce power consumption by over 99.99% [65]. Extremely long battery life is possible by keeping the CPU in sleep state for most of the time, only occasionally waking up to perform its sensing task. Since RAM requires power to maintain its state even in sleep mode, it is usually limited to only a few KB of RAM, a full six orders of magnitude less than most modern computers.

In 2001 Pister predicted that by 2010, the size of these devices would be reduced to a cubic centimetre, and cost to less than a dollar [72]. While the first prediction has come true [96], the latter so far has not. Future improvements in IC technology may allow more powerful devices at the same level of cost and power consumption, but for many applications an increase in battery lifetime or a reduction in cost may be more valuable, and may enable new applications not possible at the current level of technology.

Thus, much of the research into WSN is about the trade-offs involved in achieving useful functionality in as small a space as possible, gradually exploring the design space between capabilities, accuracy and performance on one side, and their cost in terms of memory and power consumption on the other. New protocols were developed at every layer in an application, optimising them for the specific constraints of sensor nodes. This includes lightweight MAC protocols for radio communication, trading latency for energy by turning off the radio as much as possible [103, 94], lightweight operating systems and virtual machines, trading functionality for reduced size and complexity [54, 35, 37, 52, 13], lightweight routing and data aggregation [44, 11], lightweight data compression and

reprogramming techniques, trading CPU cycles for a reduction in transmitted bits [63, 76], lightweight localisation, trading accuracy for reduced complexity [68, 78, 79], etc.

What these have in common is that they all revisit classic computer science problems, and adjust them to fit one a sensor node, making trade-offs to optimise for power consumption, either directly by reducing the time the processor or radio is active, or indirectly by reducing code size and memory consumption enough for them to run on the extremely low power, but very resource-constrained CPUs.

1.1 Internet-of-Things

Recently, research into the Internet-of-Things (IoT) focusses on connecting many everyday objects and building smart applications with them. In this vision, similar to Weiser's ubiquitous computing, any object could be connected to the internet, and cooperate to achieve useful goals. For example a house with a smart airconditioning system may use sensors in each room, weather forecast information downloaded from the internet, past data on how the house responds to weather changes, and the user's current location, and combine all this information to conserve energy while making sure the house is at a comfortable temperature when the user gets home.

While IoT and WSN overlap and the two terms are sometimes used interchangeably, an important difference is that in WSN research, applications typically consist of a large number of homogeneous and resource-constrained nodes, where IoT devices come in a wide range, with vastly different performance characteristics, cost, and power requirements.

On one end of this spectrum are devices like the Intel Edison and Raspberry Pi. These are the result of another decade of miniaturisation since the beginning of WSN research, and are basically a complete PC in a very small form factor. They are powerful enough to run a normal operating system like Linux, but relatively expensive and power hungry. On the other end are more traditional WSN CPUs like the Atmel ATmega or Texas Instruments MSP430: much less powerful, but also much cheaper and low power enough to potentially last for months or years on a single battery. Since both classes of devices have

such different characteristics, solutions that are appropriate for one usually do not work for the other.

A second important difference between WSN and IoT applications is that in WSN applications, the network is usually dedicated to a specific task and the hardware is an integral part of the design of the application. In the broadest IoT vision, the smart devices in the user's environment cooperate to implement new applications, but these devices may come from many different vendors and may not be specifically designed for the application the user wants to run. Coming back to the example from Weiser's paper [99], it is unlikely a user would be willing to buy a matching pair of a coffee maker and an alarm clock, just so that they will work together to have his coffee ready in the morning. The challenge for IoT is to allow different smart coffee makers and smart alarm clocks to be programmed in such a way to enable this application.

Thus, many IoT applications are inherently heterogeneous, and as Gu points out [35], even when powerful devices are used like the Raspberry Pi, it is not unusual for low power devices to be included to form a hybrid network and take advantage of their extremely long battery lifetime. One of the main challenges then becomes how to programme these networks of IoT devices.

1.2 Virtual machines

The use of virtual machines has been common in desktop computing for a long time, with Java and .Net being the most well-known examples. There are several advantages to using VMs, the most obvious one being platform independence. Java enables a vast number of different models of Android phones to run the same applications. In a heterogenous environment as IoT applications are expected to be, a VM can significantly ease the deployment of these applications if the same programme can be run on any node, regardless of its hardware platform. A second advantage is that a VM can offer a safe execution environment, preventing buggy or malicious code from disabling the device.

Since the early days of WSN research, many VMs, some based on Java and .Net, have been developed to run on resource-constrained sensor nodes. They manage to pack

Table 1.1: Slowdown for interpreting sensor node VMs

VM	Source	Platform	Performance vs native C
Darjeeling	Delft University of Technology	ATmega128	30x-113x slower [13]
TakaTuka	University of Freiburg	Mica2 (AVR) and JCreate (MSP)	230x slower [24]
TinyVM	Yonsei University	ATmega128	14x-72x slower [41]
DVM	UCLA	ATmega128L	108x slower [9] 555x slower [49]
SensorScheme	University of Twente	MSP430	4x-105x slower [27]

an impressive set of features on such a limited platform, but sacrifice performance, and usually do not provide a safe execution environment.

1.2.1 Performance degradation

The VMs for which we have found concrete performance data are shown in Table 1.1. The best case, the 4x slowdown seen in one of SensorScheme's benchmarks, is a tiny benchmark that only does a single call to a random number generator, so this only tells us a function call costs about 3 times longer than generating the random number. Apart from this single data point, all interpreting VM are between one and two orders of magnitude slower than native code.

In many scenarios this may not be acceptable for two reasons: for many tasks such as periodic sensing there is a hard limit on the amount of time that can be spent on each measurement, and an application may not be able to tolerate a slowdown of this magnitude. For applications that sample close to the maximum rate a node could process, *any* reduction in performance directly translates to a reduction in the sampling rate.

Perhaps more importantly, one of the main reasons for using such tiny devices is their extremely low power consumption. In many applications the CPU is expected to be in sleep mode most of the time, so little energy is spent on the CPU compared to communication or sensors. However, if the slowdown incurred by a VM means the CPU has to stay in active mode 10 to 100 times longer, this means 10 to 100 times more energy is spent on the CPU and it may suddenly become the dominant factor and reduce battery lifetime.

To illustrate this we will look at three concrete examples below.

Table 1.2: Energy consumption breakdown for the Mercury motion analysis application, source: [58]

Component	Energy (μJ)
Sampling accel	2,805
CPU (activity filter)	946
Radio listen (LPL, 4% duty cycle)	2,680
Time sync protocol (FTSP)	125
Sampling gyro	53,163
Log raw samples to flash	2,590
Read raw samples from flash	3,413
Transmit raw samples	19,958
Compute features	718
Log features to flash	34
Read features to flash	44
Transmit features	249
512-point FFT	12,920

Mercury

Few WSN or IoT applications report a detailed breakdown of their power consumption. One that does is a platform for motion analysis called Mercury [58]. The data reported in their paper is copied in Table 1.2. The greatest energy consumer is the sampling of a gyroscope, at 53,163 μJ. Only 1,664 μJ is spent in the CPU on application code for an activity recognition filter and feature extraction. When multiplied by 10 or 100 however, the CPU becomes a very significant, or even by far the largest energy consumer.

Table 1.2 also shows that transmitting raw data is a major energy consumer. To reduce this, Mercury has the option of first extracting features from the raw sensor data, and transmitting these instead, achieving a 1:60 compression. Mercury has five feature detection algorithms built in: maximum peak-to-peak amplitude; mean; RMS; peak velocity; and RMS of the jerk time series. But they note that the exact feature extractor may be customised by an application.

This is the kind of code we may want to update at a later time, where using a VM could be useful to provide safety and platform independence. However, at more than a 34x slowdown in the feature extraction algorithm, which is in the range seen for most VMs, this would be pointless, because more energy would then be spent in the CPU than we would save on transmission.

Finally, a more complex operation such as a 512 point FFT costs 12,920 mJ. For tasks like this, even a slowdown by a much smaller factor will have a significant impact on the total energy consumption.

Lossless compression

As a second example we consider lossless data compression. Since the radio is often one of the major power consumers on mobile devices and one of the main tasks of sensor nodes is collecting and transmitting data, compressing this data before it is sent is an attractive option to conserve energy spent on transmission. However, energy must also be spent on CPU cycles during (de)compression. Barr and Asanović have analysed this trade-off for five compression algorithms on the Skiff research platform, with hardware similar to the Compaq iPAQ. The break-even point was found to be at about 1000 instructions for each bit saved, beyond which the energy spent on compression would start to outweigh the energy saved on transmission. In their experiments this was the case for a number of combinations of compression algorithms and input data where compression led to an increase in total energy consumption, compared to sending uncompressed data [10].

Most of the traditional algorithms, such as the ones considered by they Barr and Asanović, are too complex to run on a sensor node, so specialised compression algorithms have been developed for sensor nodes. One such algorithm is LEC [63], a simple lossless compression algorithm that can be implemented in very little code and only needs to maintain a few bytes of state. We will use a rough calculation to show why performance is also a concern for the simpler compression algorithms developed for sensor nodes.

Using the power consumption data from the datasheets for the Atmel ATmega128 [65] CPU, we estimate the energy per CPU cycle in active mode. Running at 8 Mhz and 2.7V, the ATmega consumes 7.5 mA.

$$7.5mA * 2.7V/8MHz = 2.53nJ/cycle \tag{1.1}$$

We can do a similar calculation to determine the energy per bit for the Chipcon CC2420 IEEE 802.15.4 transceiver [16]. This radio can transmit at 250 kbps, but the small frame

Table 1.3: LEC compression energy savings

Energy consumption	
ATmega128 per cycle	2.53 nJ/cycle
CC2420 per transmitted bit	288.2 nJ/bit
LEC compression	
bits saved	2256 bits
cycles spent	97052 cycles
cycles per bit	43 cycles/bit
Energy saved	650 µJ
Energy expended	246 µJ
Ratio saved/expended	2.6x

size of the 802.15.4 protocol introduces a relatively large overhead, and Latreé et al. calculate a maximum throughput of 163 kpbs [51]. The CC2420 can also operate at 2.7 V, and consumes between 8.5 and 17.4 mA depending on transmission power. Using the higher value, so that compression will be more worthwhile, this yields

$$17.4mA * 2.7V/163kbps = 288.2nJ/bit \tag{1.2}$$

As a result, we can spend $288.2/2.53 \approx 114$ cycles per bit to reduce the size of the transmitted data, and still conserve energy using compression.

We implemented the LEC compression algorithm and used it to compress a dataset of 256 16-bit ECG measurements [71], or 4096 bits of data. The results are shown in Table 1.3. LEC compression reduced the dataset to 1840 bits, saving 2256 bits, or 651 µJ on transmitting the data, at the expense of 246 µJ extra energy spent in the CPU.

This shows that for this combination of hardware and sensor data, LEC compression is effective in reducing total energy consumption. However the energy saved is only 2.6x more than the extra energy expended on the CPU. This means that if running the compression algorithm in a VM slows it down by a factor of more than 2.6x, this would tip the balance in favour of just sending the raw data.

Where exactly this break-even point lies depends on many factors. The CPUs and radio used in this calculation are very common in typical sensor nodes, for example the widely used Telos platform [73] is based on the CC2420 radio and the ATmega CPU is found in many Arduinos. However many parameters will affect the results. Power consumption is

roughly linear in relation to clock frequency, so at the same voltage the cost per cycle will be similar, but at lower frequencies the CPU can operate at a lower voltage which lowers the cost per cycle. The cost per bit depends on many factors including the link quality. A bad link will increase the cost per bit due to retransmissions, but if the node transmits at a lower power, for example to reduce the number of neighbours and collisions, the cost per bit will be lower than calculated.

This calculation is another example of a situation where the slowdown caused by current sensor node VMs means compression is not worthwhile. For Mercury's feature extraction, the break-even point is around 27x slowdown, while for this case of LEC compression it is at only 2.6x. The exact numbers depend on the application, but it is clear that a slowdown of one to two orders of magnitude will affect battery lifetime in many applications.

Amulet

A final example to motivate both the usefulness of a VM and the need for good performance is the smart watch platform Amulet [40]. Amulet aims to provide a week-long or month-long battery life time. To do so it uses a typical low-power resource-constrained CPU, the MSP430.

Amulet supports multiple concurrent applications. Currently these are written in Amulet C, and are compiled together into a single firmware image that is loaded onto a watch. However, the authors envision a future with multiple vendors of Amulet devices, who will no doubt use different hardware platforms, and many developers submitting applications to a sort of app store. In such a scenario the platform independence a VM can provide is a valuable property.

One of the main design goals of Amulet is long battery life, and although the authors do not provide a breakdown of energy consumption per component like Mercury, they do note that "Energy consumption is significantly impacted by the fraction of time the application microcontroller (MSP430) is active". This suggests a large performance overhead would also significantly reduce battery lifetime.

Table 1.4: Sleep and active time for Amulet applications, source: [40]

Application	%Sleep	%OS	%App
Clock	98.1	0.9	1.0
EMA	98.2	1.0	0.8
Heart rate	91.1	0.9	8.0
Pedometer	93.8	2.2	4.0
Pedometer+HR	87.5	1.9	10.6
Pedometer+HR+Clock	85.4	2.8	11.8

The paper provides an overview of the percentage of time spent in sleep mode, in the OS, or in application code for a few different applications. This is reproduced in Table 1.4. The percentage of time the CPU is executing application code varies between 0.8% and 11.8%. Combined with the one to two orders of magnitude slowdown seen in Table 1.1, there are many cases where the slowdown of a VM would mean the CPU has to be active for more than 100% of the time, indicating it does not have enough time to complete all its tasks in time.

For the most expensive configuration that spends 11.8% executing application code, this happens at a slowdown of roughly 8.5x. The lighter applications can tolerate more overhead, but the highest slowdowns seen in Table 1.1 are a problem for even the lightest application.

Ahead-of-Time compilation

Thus, a better performing VM is needed, preferably one that performs as close to native performance as possible. Translating bytecode to native code is a common technique to improve performance in desktop VMs. Translation can occur at three moments: offline, ahead-of-time (AOT), or just-in-time (JIT).

JIT compilers translate only the necessary parts of bytecode at run time, just before they are executed. They are common on desktops and on more powerful mobile devices, but are impractical on sensor node platforms, some of which can only execute code from flash memory. This means a JIT compiler would have to write to flash memory at run time, which is expensive and would cause unacceptable delays. There are nodes that can execute code from RAM, but the small amount of it present on sensor nodes means a JIT

compiler would either have to allocate a large part of this scarce resource to the compiled code cache, or frequently recompile the same methods if they get flushed after the cache overflows [24].

Translating to native code offline, before it is sent to the node, has the advantage that more resources are available for the compilation process. We do not have a Java compiler that compiles to our sensor node's native code to test the resulting performance, but we would expect it would come close to compiled C code in many cases. However, doing so, even if only for small, performance critical sections of code, sacrifices the two of the key advantages of using a VM: The host now needs knowledge of the target platform, and needs to prepare a different binary for each type of CPU used in the network, and for the node it will be difficult to provide a safe execution environment when it receives binary code.

Therefore, this dissertation will focus on the middle option: translating the bytecode to native code on the node itself, at load time. We will build on previous work by Joshua Ellul [24] on AOT translation on sensor nodes. This approach reduces performance overhead to a slowdown of up to 9.1x, significantly faster than the interpreting VMs, but not fast enough for LEC compression to be worthwhile in our example. Unfortunately, it also results in an increase in the size of the stored programmes of up to 4.5x, which limits the size of the programmes that can be loaded on a node.

1.2.2 Safety

Low-cost low-power sensor node CPUs have a very simple architecture. They typically do not have a memory management unit (MMU) or privileged execution modes to isolate processes. Instead, the entire address range is accessible from any part of the code running on the device.

At the same time, sensor node code can be quite complex. While programming in a high-level language can reduce the risk of programming errors, the limited resources on a sensor device still often force us to use more low-level approaches to fit as much functionality and data on a device. For example by storing data in simple byte arrays

instead of using more expensive objects, or a few cases where we have had to explicitly set a variable to `null` to allow an object to be garbage collected earlier than it otherwise would have been. In such an environment, mistakes are easily made, and with full access to the entire address space can have catastrophic consequences. A second threat comes from malicious code. As IoT applications become more widespread, so do the attacks against them, and the unprotected execution environment of sensor node CPUs makes them an attractive target.

To guard against both buggy code and malicious attacks, a desirable property would be the ability to execute code in a sand boxed manner to isolate untrusted application code from the VM itself. Specifically, we would like to guarantee that malicious code cannot:

1. Write to memory outside the range assigned by the VM.
2. Perform actions it does not have permission for.
3. Retain control of the CPU indefinitely.

Note that these guarantees do not assure the correctness of the application itself: buggy code may still corrupt its own state. More fine-grained checks can be useful to reduce the risk of bugs and speed up the development process by detecting them earlier. Safe TinyOS [19] adds run-time checks to detect illegal writes, and can do so efficiently by analysing the source code before it is compiled. However, this does not protect against malicious code being sent to the device and depends on the correctness of the host.

Our approach depends only on the correctness of the VM, and guarantees it can always regain control of the node and terminate any misbehaving application before it executes an illegal write or performs an action it is not permitted to.

Amulet

The Amulet [40] smart watch platform is also a good motivating example for the need for a safe execution environment. Since it aims to run multiple concurrent applications possibly developed by different developers, it is important to isolate these applications from each other and from the OS.

Amulet does this by using a restricted dialect of C, Amulet C, which has several limitations that make it easier to guarantee safety. For example, there is no access to arbitrary memory location through pointers, no goto statements, no dynamic memory allocation, and no recursion. The Amulet compiler then adds runtime checks where static safety checks are not sufficient.

Although the authors do not provide any data on the overhead caused by the restrictions in Amulet C and the added run time checks, it is significant enough to motivate them to investigate the use of memory protection units found on some recent CPUs [39] to reduce this overhead. However these MPUs are only available on a limited number of CPUs.

To use a VM in a project such as Amulet, it must be able to provide the same level of isolation, and do so at an acceptable cost.

1.3 Scope

Internet-of-Things devices come in a wide range with varying capabilities. The larger IoT platforms are powerful enough to run standard operating systems, tools and languages, and VMs are well established as a good way to programme devices powerful enough to run advanced JIT compilers. This dissertation focusses specifically on small sensor nodes for which no such standards exist. These platforms have the following characteristics:

- Separate data and programme memory: Memory is split into RAM for data, and flash memory for code. While some device can execute code from both RAM and flash [87], others cannot [65].

- Very limited memory: Since volatile memory consumes energy even when the CPU is in sleep mode, it is typically restricted to 10 KB of RAM or less. More non-volatile flash memory is available to hold programmes, but at 32 to 256 KB this is still very limited.

- Low complexity CPUs: While usually rich in IO to drive actuators and read from sensors, the rest of the CPUs used in these devices is very simple in design to save cost and power consumption. Instructions usually take a fixed number of cycles,

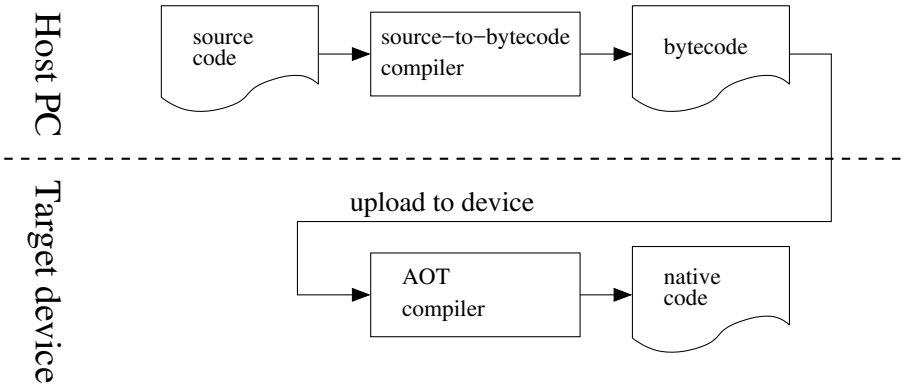

Figure 1.1: Highlevel overview of the compilation process

since memory is on chip access times are constant and fast, and there are no complicating factors like deep pipelines, caches, or branch predictors.

- Very limited energy budget: Typical usage scenarios demand a long battery lifetime, since frequent replacement or recharging would be too impractical or costly. All aspects of WSN software design are therefore focused towards minimizing energy use.

Our focus on sensor nodes raises the question of how platform independent our VM is, since IoT applications may mix both classes, and the ability to use a single binary to programme multiple classes of devices is exactly one of the advantages of using a VM. While the optimisations we propose are specifically developed to work in a resource-constrained sensor device, the bytecode is very close to standard Java, so a more powerful device would have no problem running it just as efficiently as it runs normal Java code.

AOT compiler Figure 1.1 shows the high level process from source code to native code running on the node. The host PC will compile source code to bytecode, which is uploaded to the target node. Instead of interpreting this bytecode, our VM will translate it at load time and store the resulting native code in flash, which is then executed.

We will see that in order to achieve good performance, an optimising source-to-bytecode compiler is necessary to generate high quality bytecode. However, the focus in this dissertation is on the AOT compiler running on the sensor node, and on what performance it can deliver, given good quality bytecode.

Although we will make some changes to the way the bytecode is produced, building a full optimising source-to-bytecode compiler is outside the scope of this dissertation. To determine what level of performance is possible, some manual optimisations are applied to the source code to produce better quality bytecode, most of which an optimising compiler could be expected to do automatically.

Source language Note that in Figure 1.1 we use 'source code' and 'bytecode' instead of Java or JVM bytecode, since the use of Java was only motivated by the availability of existing work to build upon, most notably the Darjeeling VM [13], not because we believe Java to be a particularly good choice. The techniques described in this dissertation are not specific to Java, but can be applied to any language that compiles to a stack based bytecode.

We will see that in fact Java, in its current form, is not well suited for sensor node applications, and we will end this dissertation with a number of suggestions on how to remedy some of these issues.

1.4 Research questions and contributions

There are clear advantages from using virtual machines, especially in heterogeneous and dynamic IoT scenarios, but these come at a cost. On a sensor node, this cost is especially significant since they are already very resource-constrained and cannot do many of the optimisations used in VMs on larger devices.

The main research question of this dissertation is whether virtual machines are a suitable means to programme sensor node devices from three perspectives. Specifically:

a. Performance: How close can an Ahead-of-Time compiling sensor node VM come to native C performance, and what are the trade-offs?

b. Safety: Can a VM be an efficient way to provide a safe, sandboxed execution environment on a sensor node?

c. Language: Is Java a suitable language for a sensor node VM, and how may it be improved?

This dissertation makes the following contributions:

1. We identify the major sources of overhead when using the approach to Ahead-of-Time compilation described by Ellul and Martinez [25, 24].

2. Using the results of this analysis, we propose a set of eleven optimisations to address each source of overhead. These include improvements to Ellul's AOT approach, modifications to the VM's bytecode, and a lightweight alternative to standard Java method invocation.

3. We show that in addition to these improvements to the AOT compiler, better optimisation in the Java to JVM bytecode compiler is critical to achieving good performance.

4. We describe a number of checks that are sufficient for the VM to provide a safe, sandboxed execution environment, and show most checks can be done at load time, reducing the overhead of run-time checks.

5. We evaluate our optimisations using a set of benchmarks with varying characteristics, including the commonly used *CoreMark* benchmark [88] and a number of real sensor node applications. We show these optimisations:

    - Reduce the size of the generated code by 40%, allowing larger programmes to be loaded, and quickly compensating for the increase in VM size due to these optimisation,
    - Allow constant data to be placed in flash memory, enabling applications with constant tables that could otherwise not be implemented,
    - Eliminate 91% of the performance overhead caused by the VM's stack-based architecture, and 79% of performance overhead overall.

6. Using the same benchmarks we evaluate the cost of our safety checks, and show this cost to be comparable to, or lower than the two existing native code approaches to provide a safe execution environment on a sensor node, while providing platform independence at the same time.

7. Finally, we identify a number of aspects of the Java language and virtual machine that ultimately make it a bad match for sensor nodes, and propose ways to address these issues in future sensor node VMs.

1.5 Structure of thesis

Chapter 2 introduces necessary background knowledge on wireless sensor networks, Java and the Java virtual machine, and AOT compilation.

Chapter 3 discusses the state of the art in improving performance for sensor node VMs and providing safe execution environments.

Chapter 4 describes the global design of CapeVM.

Chapter 5 first analyses the sources of overhead for the current state of the art in sensor node AOT compilers, and then presents a set of optimisations to address each of these

sources. Where there were multiple options to implement an optimisation, we describe alternatives and motivate our choice.

Chapter 6 presents a set of checks that allow CapeVM to provide the sandbox safety guarantees described in Section 1.2.2, and shows how the JVM's simple design allows for most of these checks to be done at load time.

Chapter 7 evaluates the effect of the optimisations presented in Section 5 and the cost of the safety checks presented in Section 6. We use a set of benchmarks with different characteristics, both small benchmarks to show the behaviour in extreme cases and real sensor node applications.

Chapter 8 describes a number of issues we encountered while doing this work, which show standard Java is not the best choice for a sensor node VM, and suggests ways to improve these in future sensor node VMs.

Chapter 9 concludes this work.

1.6 List of publications

Part of the work presented in this dissertation is based on previously published reports or conference proceedings:

- N. Reijers, K. J. Lin, Y. C. Wang, C. S. Shih, and J. Y. Hsu, "Design of an Intelligent Middleware for Flexible Sensor Configuration in M2M Systems", 2nd International Conference on Sensor Networks (SENSORNETS), February 2013.

- N. Reijers and C.-S. Shih, "Ahead-of-Time Compilation of Stack-Based JVM Bytecode on Resource-Constrained Devices", International Conference on Embedded Wireless Systems and Networks (EWSN), February 2017.

- N. Reijers and C.-S. Shih, "Improved Ahead-of-Time Compilation of Stack-Based JVM Bytecode on Resource-Constrained Devices", arXiv:1712.05590, December 2017.

1.7 Naming

In literature various names are used for our target devices. The word *mote* was common in early research on wireless sensor networks, as was *sensor node*, or simply *device*, although the latter is also used for larger, more powerful devices. In this dissertation we will use the terms *sensor node* or just *node* interchangeably to refer solely to the type of severely resource-constrained devices used in both WSN and IoT applications.

When reprogramming these nodes, a new programme must be sent to them by a more powerful device. This role is refered to in literature by various names, including *server*, *gateway*, *master*, *controller*, or *host*, depending on the exact design of the network and the way it is reconfigured. For the work in this dissertation these differences are not relevant, and we will use *host* to refer to the source of the code that is uploaded to the sensor node, which is assumed to be a more powerful device with desktop-class processing capabilities.

Since our VM is an Ahead-of-Time compiler, the Java source code is transformed into native code in two steps. We will use *compile time* to refer to the compilation of Java code into Java bytecode on the host, and *translation time* to refer to the translation of this bytecode into native code on the device.

Finally, we follow Dalvik in naming our virtual machine after a coastal town. In this case the beautiful city of Cape Town, where parts of CapeVM were developed over the course of two trips.

Figure 1.2: Cape Town workplace

Chapter 2

Background

This chapter will introduce some necessary background knowledge on the target hardware platform, Java and the Java Virtual Machine, and JIT and AOT compilation.

2.1 Wireless Sensor Networks and the Internet of Things

Both Wireless Sensor Networks and Internet of Things are relatively new research areas. Both consider networks of connected devices that have to cooperate to achieve some goal. There is a large overlap between the two, but there are key differences.

Wireless sensor networks is commonly understood to refer to networks of very resource-constrained devices. They are usually homogeneous, dedicated to a specific application, and in many cases battery powered. On the other hand, Internet of Things applications may contain the same class of resource-constrained devices but mix in more powerful ones as well. They may have a combination of battery- and mains powered devices, and the devices used may be a combination of devices dedicated to a particular task and smart devices that happen to be in the user's environment. One of the main challenges of IoT research is to develop ways to use the capabilities that are present in the smart devices around us to build new and useful applications.

It is this combination of heterogenous devices, and the need to reprogramme them to run new tasks that were not part of the original programming, that makes the platform independence and safe execution environment offered by a VM an attractive option.

While there is a wide range of IoT devices, they can be roughly divided in two categories. We will describe the capabilities and limitations of each of them below.

2.1.1 High-end IoT devices

Another decade of miniaturisation since the start of WSN research has allowed us to scale down devices capable of running a normal OS stack, to the size of a few centimetres. Some of the most popular examples include the Raspberry Pi range, with the Raspberry Pi Zero measuring only 65x30 mm, and the Intel Edison at 35.5x25 mm.

These devices have capabilities similar to that of desktop PCs only a few generations ago. They can run a normal operating system like Linux and all the standard protocols and tools that come with it. Compared to the traditional resource-constrained sensor nodes, they can perform much more complex tasks, but the smallest devices in this class are still significantly larger than sensor nodes, more expensive, and most importantly, consume significantly more power.

Tung et al. [91] report measurements on the Intel Edison, one of the most low-power devices in its class, showing an active power consumption of up to 130 mA and sleep power consumption of 1.8 mA, at 3.7 V. Additionally, the wake-up response time was measured around 380 ms, while the resource-constrained CPUs described below can wake up in only a few cycles or ms, depending on their configuration.

Since devices in this class are capable of running normal, well established VMs, we do not consider them in this dissertation, but instead focus on the second class of devices: sensor nodes.

2.1.2 Resource-constrained sensor nodes

The second class of devices, wireless sensor nodes, are distinctly less powerful. They are designed to be deployed at low cost and potentially in large numbers, and to be capable of running for weeks or months on a single battery charge. As a typical example, the MICAz node [21] uses only 30mA when active and 16µA in sleep mode. More recently, the Arduino family of devices, based on similar hardware, has led to a very active community

Table 2.1: Main characteristics of the ATmega128 and MSP430F1611 CPUs

	ATmega128 [65, 64]	MSP430F1611 [87, 86]
Number of registers	32	12
Register size	8-bit	16-bit
RAM	4 KB	10 KB
Flash	128 KB	48 KB
Frequency	up to 16 MHz	up to 8 MHz
Simple instruction cost	1 cycle	1 cycle
Memory access cost	2 cycles	2 to 6 cycles
Branch cost (taken/non-taken)	2 / 1 cycles	2 cycles
Active power consumption[a]	7.5 mA	4.3 mA
Deep sleep power consumption	0.3 µA	0.2 µA
Can execute code from	Flash	Flash and RAM

[a] at 8MHz and 2.7V

of both research and hobby projects.

While an enormous number of different hardware platforms have been developed, the components they use come from a limited set. Two of the most popular families of CPUs used in these platforms are the Atmel AVR and Texas Instruments MSP430. Both families of CPUs come in a large number of variations with different amounts of memory, IO ports, and physical packages, but the underlying architecture is similar for all. Table 2.1 lists the main characteristics for two popular members for both families, the ATmega128 and the MSP430F1611. Below we will describe the most important properties for this class of devices that are relevant to the work in this dissertation.

Memory Memory is split into persistent flash memory for code, and volatile RAM for data. The MSP430 CPUs have a von Neumann architecture and can execute code from both, while the AVR's Harvard architecture can only execute code from flash memory. Flash memory is typically in the range of 16 KB to 256 KB, while RAM, which consumes energy even in sleep mode, is restricted to up to 10 KB. There are no caches, and since both memories are on chip, access times are constant and take only a few cycles.

Simple architecture These CPUs achieve their extremely low cost and power consumption by restricting themselves to a very simple design. Each instruction takes a fixed number of cycles, varying only for taken or non-taken branches. Since most instructions only take one or two cycles, there is no pipelining or need for branch prediction. There is also

no memory management unit or protection rings, and no floating point support.

Operating system The severe resource restrictions on these devices mean that a normal layered architecture with an OS, networking stack, and applications running on top of that is not possible. The closest thing to a widely accepted OS for sensor nodes is TinyOS [54] which provides several basic services for IO, communication and task management. Contrary to a normal OS, TinyOS does not 'load' an application, but is statically linked with the application's code to form a single binary which is then programmed into the node's flash memory. Thus, sensor node applications are often a single binary, running directly on the CPU.

Several systems exist that allow over the air reprogramming of sensor nodes [75]. In some cases these allow the entire application code to be replaced, including the reprogramming protocol itself [76]. In other cases the reprogramming system may be permanent and include other basic services, more closely resembling an operating system. But even in these cases the restricted amount of flash memory means that such a system cannot afford to waste large amounts of memory on library functions that may never be used by the application, so the services provided by such an 'operating system' are quite restricted, leaving much of the low level work to the application.

A number of sensor node virtual machines have been developed that allow the application to be updated remotely. These obviously provide a higher level of abstraction from the underlying hardware. However, it is important to note here that these virtual machines are not an extra layer, between a lightweight OS and the application, but often *replace* the OS entirely, so the VM runs directly on the CPU. This kind of cross layer optimisation, or complete merging of layers is typical of many sensor networks.

2.2 The Java virtual machine

Next, we will briefly introduce the Java virtual machine, and describe some details relevant to this work.

The first public release of Java was in 1995. It consists of two separate but closely

related parts: the Java programming language, and the Java virtual machine (JVM): an abstract machine specification, running programmes written in JVM bytecode. Since the release of Java, several other languages have been developed that compile to JVM bytecode and can run on the same virtual machine.

It was quickly adopted by web browsers to run interactive *applets*. Two key properties of Java contributed to this success:

- Implementations of the JVM were built for many hardware platforms, so the same applet could be run in any browser, regardless of the hardware it was running on.
- It allowed users to safely run applets from untrusted sources since the virtual machine runs them in a 'sandboxed' environment with access to only those system resources explicitly allowed by the user.

For stand alone desktop applications Java also became popular because it was an easy to learn, object oriented, garbage-collected language that allowed for a higher level of programming than C or C++, all of which boosted developer productivity.

2.2.1 JVM bytecode

Compared to other widespread desktop virtual machines such as Lua, Python and .Net, Java's bytecode is very simple. Java is a *stack-based* virtual machine, as opposed to a *register-based* virtual machine: almost all operations take their operands from an operand stack, and push their results back onto it. For example, Listing 1 shows how the statement `a=b+c;` may be translated into JVM bytecode. First, `b` and `c` are loaded onto the stack, the `IADD` instruction then pops these operands from the operand stack and pushes the sum back onto it, and finally `ISTORE` stores the result back into a local variable.

```
1  //JVM instruction     // JVM stack
2
3  ILOAD_1              b
4  ILOAD_2              b   ,   c
5  IADD                 b+c
6  ISTORE_0
```

Listing 1: JVM bytecode for `a=b+c;`

By far the largest number of instructions, 99 out of 206, are for loading or storing data to and from the operand stack. These come in different flavours for different datatypes: `ILOAD` loads an int, while `BLOAD` loads a byte onto the stack. 53 instructions are simple arithmetic or bitwise operations, such as `IADD` in the example, also in different variations for different datatypes. There are 39 instructions for branches and method invocations, and 15 for various other tasks such as creating new objects and throwing exceptions.

Each JVM bytecode is encoded as a single byte. Some are followed by one or more operands, for example the method to call, or the type of object to create, but most are not. This very simple instruction set makes it a good match for a resource-constrained sensor node.

2.2.2 Memory

The JVM stores information in three different places:

- The stack frame: each method's stack frame contains a section for its operand stack, and its local variables.
- Global variables: static variables that are allocated globally when a class is loaded (we ignore `ThreadLocal` variables since they are not supported in our VM).
- The heap: objects and arrays are stored here, and automatically garbage-collected when no longer used.

The JVM is a 32-bit machine. All the places where data may be stored, objects on the heap, operand stacks, local variables, and a class' static variables, are blocks of 32-bit wide slots. 64-bit **`long`** and **`double`** types occupy two slots, while the shorter **`byte`**, and **`short`** types are sign-extended and stored as a 32-bit value.

Figure 2.1 shows a graphical representation of this. An important difference with languages such as C, Pascal or C# is that in JVM the only value types are the various integer types, and references. There are no compound types like a C **`struct`** or Pascal **`record`**. Object live in heap, and only there, and the operand stack, and local, static or instance variables only contain references to objects.

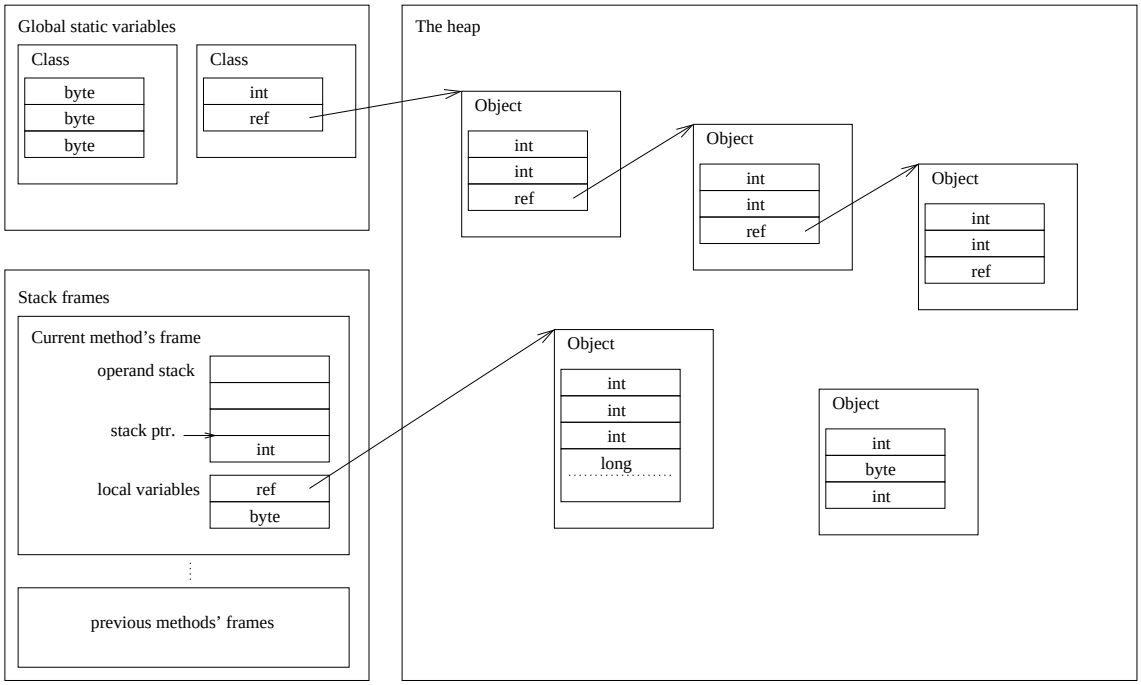

Figure 2.1: Highlevel overview of JVM memory design, and some example data structures

2.2.3 Sandbox

A sandbox is a security mechanism for isolating a process from the environment in which it runs. They can be used to run code from untrusted sources, without risk of harm to the host machine or other applications running on it.

In the JVM's case, programmes are written to run on the abstract JVM machine model. Besides providing platform independence, this also means JVM programmes have no knowledge of the hardware platform they are running on. All communication with the outside world happens through the Java standard library classes implemented by the JVM in native code, which gives the virtual machine firm control over the resources an application may access.

In addition, the JVM will verify the bytecode at load time to make sure it is well formed and adheres to the JVM standard [56]. It performs many checks, for example that branches are within the bounds of the method, and branch to the beginning of an instruction, that execution cannot fall off the end of a method, that no instruction can access or modify a local variable at an index greater than or equal to the number of local variables that its method indicates it allocates, that the exact operand stack depth and the type of values on

the stack is known at any point and does not over- or underflow, etc.

2.2.4 WAT, AOT, and JIT compilation

While the popularity of Java rose quickly after its introduction, it also very quickly got a reputation for being slow. As Tyma put it in 1998 ”The plain truth is: Java is slow. Java isn’t just slow, it’s *really* slow, *surprisingly* slow. It is ’you get to watch the buttons being drawn on your toolbar’ slow.” [92].

The main reason is all early implementations of the JVM were interpreters. An interpreter executes a programme by retrieving instructions from memory one at a time, and then executing them. An outline of what a typical interpreter’s main loop looks like is shown in Listing 2. For each instruction, the VM needs to (i) retrieve the bytecode at the current programme counter, (ii) increment the programme counter, (iii) jump to the correct case label, (iv) execute the instruction, and (v) loop for the next iteration.

```
1    while (true) {
2        opcode = bytecode[pc];
3        pc++;
4        switch (opcode) {
5            case ILOAD_0: ...
6            case ILOAD_1: ...
7            ...
8    }
```

Listing 2: Outline of a typical interpreter loop

Since most instructions are very simple, for example simply adding two operands, the relative overhead from these steps is very high. Interpreters spend most of their time on the interpreter loop, and only a fraction of the time on actually executing JVM instructions.

Thus, a common approach to improve JVM performance is to translate the bytecode to the native machine code of the target platform before executing it. Three main approaches exists, which differ the point at which the bytecode is translated to native code.

Compile time Borrowing the term from Proebsting et al., Way-Ahead-of-Time (WAT) compilers translate to native code during or directly after compiling the Java sources [74]. Some systems first translate to C [23], which is then compiled using normal optimising C compilers.

Regardless of which approach is chosen, the result is a native binary for the target platform, rather than JVM bytecode. The advantage of this approach is that ample time and resources are available at compilation time so highly optimised code can be produced. However, the downside is that the resulting code is no longer platform independent or guaranteed to be properly sandboxed.

Load time A second group of compilers translate bytecode to native code at load time. In these cases the entire application is translated to native code, before it is run. Therefore, they are usually called Ahead-of-Time (AOT) compilers. An example of this approach are early versions of Android's ART runtime, which translate an app completely, at the moment it is downloaded onto a device (although it since has mixed in JIT techniques as well, discussed below).

The advantage is that it combines the advantage of WAT, being able to spend considerable resources on optimisation, with platform independence and a guaranteed sandbox, since the translation is now fully under control of the device running the application, rather than the device that compiled it. A downside is that the initial translation adds to the time it takes to load or install an application.

Run time Finally, the last group, Just-In-Time (JIT) compilers, incrementally translate the bytecode to native code while the application is running. While an obvious downside is that this may initially slow down the application while it is translating bytecode at run time, a JIT compiler can take advantage of the observed run-time characteristics to make better optimisation decisions, or do more aggressive optimisations that may have to be rolled back if some preconditions no longer hold, for example inlining a virtual method as long as only a single implementation is loaded [45].

Chapter 3

State of the art

This chapter presents the state of the art in Internet of Things and sensor networks relevant to the work in this dissertation. It starts with existing work on programming WSN and IoT networks, and the virtual machines developed for them. Next, it discusses proposed ways to improve sensor node VM performance, and ways to guarantee safety on sensor devices.

3.1 Programming WSN and IoT devices

This challenge of programming IoT devices can be split into two questions:

- How can we build applications at a higher level, coordinating the behaviour of many devices without having to specify the behaviour from each device's individual perspective?
- How can we best reprogramme individual nodes safely and efficiently to support these applications?

This dissertation is concerned with the second question, and argues that a virtual machine can be an attractive option in many scenarios. But we will first discuss the higher level question of how to programme WSN/IoT applications and use one of these systems as a motivating example.

Initially, many WSN applications were built directly on top of the hardware or on some minimal operating system, such as TinyOS [54]. This results in applications being

programmed from the individual node's perspective, rather than allowing the developer to express globally what he wants from the sensor network, which makes it hard to reason about the global behaviour, especially as the number of devices and tasks increases.

Therefore, systems have been developed that make it easier for the developer to control the potentially larger number of heterogeneous nodes. Some of these are centralised, where the initiative of the application is with some central host, and devices are loaded with a runtime that allows the host to control them. Other systems are more distributed, where the application is split into components that are deployed onto nodes and from there operate more autonomously, only to receive further guidance from the host where necessary.

In the first category fall systems like sMap [22], which provides an attractive and flexible RESTful interface to a sensor network through which we can discover what sensors are available at a certain node and get or set several configuration properties. Although the authors succeeded in dramatically reducing the footprint, it is still relatively resource intensive. It is also limited in the number of properties it exposes, but the idea could easily be expanded. Similarly TinyDB [61] also makes an entire network of sensor nodes available through a central interface, in this case a SQL-like query language.

ADAE [15], developed at the IT University of Copenhagen, configures the network according to a policy describing the desired data quality, including fallback options which the system may use in case the ideal situation cannot be achieved. ADAE then dynamically reconfigures the network in response to changing conditions such as node failures, unexpected power drops, or interesting events detected by the network. However, the language used to describe the policy and constraints is hard to use and it is likely a skilled engineer is needed to translate the user's requirements into the constraint optimisation problem ADAE uses as input.

While in the previous systems applications run on a centralised host and simply control the nodes in the network, in Agilla [28] programmes are more distributed and consists of software agents that can move around in the network autonomously. While this allows some behaviours to be expressed in a natural way, the paradigm is very different from

conventional languages, and the assembly-like instruction set based on the Maté VM [52] discussed below makes it hard to use.

Cornell's MagnetOS [57], proposes a novel programming model which allows the user to write the application as a single Java application, not explicitly related to individual nodes. The system then automatically partitions the applications into pieces (by default along object boundaries), and places these pieces on nodes in the network in such a way as to minimise energy consumption. However, it requires nodes significantly more powerful than what we expect to find in a typical WSN.

LooCI [43] is a component infrastructure middleware for WSNs with standard support for run-time reconfiguration. The LooCI component model supports dynamic binding and interoperability between different hardware platforms, in their case a typical sensor node and the slightly more powerful Sun SPOT, and different languages, with implementations in C and Java. LooCI defines a list of requirements for WSN middleware, which includes supporting run-time reconfiguration to respond to changing environmental conditions, supporting heterogeneous sets of hardware, good performance, and minimal memory consumption. LooCI currently uses C to programme the smallest devices and Java for the more powerful Sun SPOT. A fast sensor node virtual machine would be a useful addition to allow more flexible and platform independent reprogramming of the smallest devices as well.

3.2 WuKong

A final example that we will look at in more detail at is WuKong [77, 55].

Applications in WuKong are written in the form of a flow based programme, an example of which is shown in Figure 3.1. Components are called WuObjects, and are instances of WuClasses. Each WuClass defines a number of input and output properties. For example, the Temperature Sensor WuClass has a single 'output' property, while the Heater has a single 'on/off' property.

When deploying an application, the host, called *master* in WuKong, will first discover the available resources in the network and then try to find a node to deploy each component

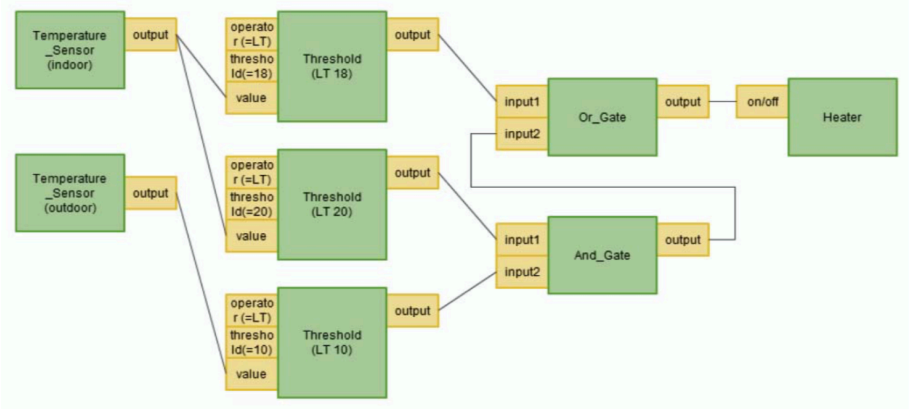

Figure 3.1: Example WuKong flow based programme

to.

A node creates WuObjects to represent its hardware components at startup, so the master can discover the sensors and actuators available in the network. When deploying the application, the master may use a hardware WuObject by connecting its properties. For software components, like the Threshold or And_Gate in the example, the master may create new instances on a node.

An WuClass is defined by:

- Its list of properties.
- A `setup()` function, called once when an object is created.
- A `update()` function, called (i) periodically, for example to sample a sensor, or (ii) when one of the input properties changes value, to compute a new output value or drive an actuator.

The `setup()` and `update()` functions can be written in either C or Java. A node may have native C implementations built in for a number of commonly used classes from the WuKong standard library, which it will advertise during the discovery phase. If the master cannot find a native implementation of a required class, it may use a Java version instead, which is slower, but more flexible since it can be deployed as part of the application. To support this, the WuKong middleware contains a version of the Darjeeling JVM [13].

The WuKong middleware takes care of propagating property changes along the links drawn in the FBP. For example, if in Figure 3.1 the output of the Or_Gate changes, the new output value is automatically propagated to the Heater's on/off property, and the Heater's `update()` function is called. An application in WuKong is defined by a number of compact tables, describing the components and links between them, and optionally a number of Java WuClasses.

In WuKong's vision, the master dynamically manages the network. A node may be used in multiple applications, and its tasks may change while the application is running if the master decides to reconfigure the network, for example in response to failure [83] or because network conditions change.

3.3 Sensor node virtual machines

Many VMs have been proposed that are small enough to fit on a resource-constrained sensor node. They can be divided into two categories: generic VMs and application-specific VMs, or ASVMs [53] that provide specialised instructions for a specific problem domain.

As an example, designed specifically for wireless sensor networks, Maté [52] was one of the first to prove sensor nodes can run a virtual machine. It provides single instructions for tasks that are common on a sensor node, so programmes can be very short.

SwissQM [67] is a more traditional and more powerful VM, based on a subset of the Java VM, but extended with sensor network specific instructions to access sensors and do data aggregation. In both systems however, the application has to be programmed in very low level, assembly-like language, limiting their target audience.

VM* [47] sits halfway between the generic and ASVM approach. It is a Java VM that can be extended with new features according to application requirements. Unfortunately, it is closed source.

Several generic VMs for high level languages like Python, LISP, and Java have also been developed, which fit on severely resource-constrained nodes. Almost all rewrite the original bytecode to their own format and employ various techniques to reduce code size.

Some functionality is sacrificed in order to fit on the sensor nodes, for instance reflection or floating point datatypes are typically not supported.

The Python-on-a-chip project [2] developed a Python bytecode interpreter small enough to fit on sensor nodes. Requiring 55 KB of flash memory and a recommended 8 KB of RAM, it fits on many, but not all of the current sensor boards. MicroPython [31] requires slightly more hardware at 256 KB flash memory and 16 KB RAM, but has its own hardware platform and runs Python 3.

LISP is one of the first high-level languages, developed at a time when room-sized computers had only slightly more capabilities than current sensor nodes. SensorScheme [27] implements a fully functioning LISP interpreter in under 41 KB, and is one of the few sensor node VMs to provide a safe execution environment.

CILIX [46] is a small VM for the .Net Common Intermediate Language implemented on the MSP430. MoteRunner [14] runs on similar devices. Instead of implementing an existing VM it targets all strictly typed programming languages and supports both Java and C#.

The smallest official Java standard is the Connected Device Limited Configuration [69], but since it targets devices with at least a 16 or 32-bit CPU and 160-512 KB of flash memory available, it is still too large for most sensor nodes. As a result, a number of Java VMs have been developed for sensor nodes, all offering some subset of the standard Java functionality, occupying different points in the trade-off between the features they provide, and the resources they require.

Darjeeling [13] from Delft University of Technology runs a modified Java bytecode with only minor restrictions, and supports multiple platforms. Similarly, TakaTuka [8] also runs on both AVR and MSP based sensor nodes, and includes its own debugger. A unique property is TakaTuka's ability to reduce garbage collection cost by static code analysis. Whenever the compiler can determine an object can be safely discarded at a certain point, it annotates the bytecode to tell the VM to do so, thus freeing up memory earlier and reducing the number of times the garbage collector has to run.

Sun's Squawk VM [80] is significantly larger than Darjeeling and TakaTuka, requiring

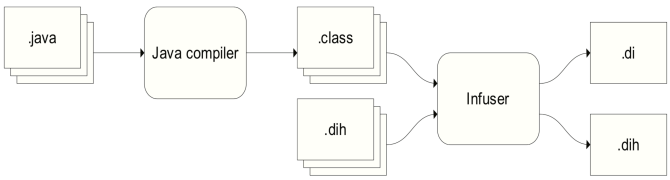

Figure 3.2: Darjeeling infusion process, source: [13]

at least 160 KB of programme memory, but offers full CLDC compliance. The Java Card VM [70] runs on more typical sensor node hardware and achieves this by also switching to a 16-bit architecture, dropping support for various features, and modifying the instructionset. At the smallest end of the spectrum is NanoVM [38], which takes a minimal approach, trading support for a large number of java opcodes for simplicity and code size. The whole VM fits in the 8K flash of an ATmega8 CPU, leaving 512bytes of EEPROM and 75% of the CPU's 1K RAM to the application.

3.4 Darjeeling

Since our VM is based on Darjeeling, we will examine it in more detail in this section.

Split VM architecture Like most other sensor node JVMs, it uses a *split VM architecture* [82]. The virtual machine running on the node does not use standard JVM class files, but these class files are first transformed by an offline tool into a format more suitable for a sensor node. In Darjeeling's case this tool is called the *infuser*, which takes several Java classes and statically links them into a single *infusion*.

The infuser changes the bytecode in several ways, replacing named references by a numbering scheme, so that the constant strings with class and method names can be removed from the constant pool, and statically linking the Java classes into a flattened list entities. Infusions are typically libraries, such as the `java.lang` base library, networking protocols, or the application. An infusion can reference code in another infusion using header files, created during the infusion process that allow the infuser to find the numbered identifiers of classes and methods in referenced infusions. This is shown in Figure 3.2.

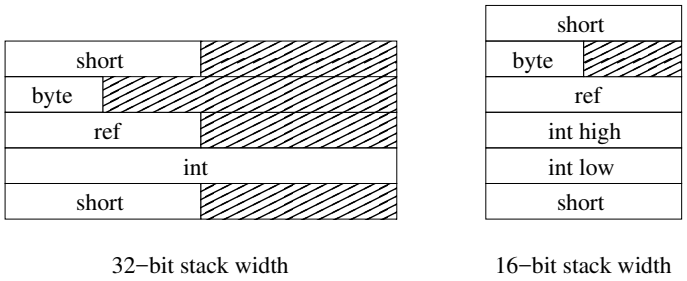

Figure 3.3: Unused memory for 32 and 16-bit slot width

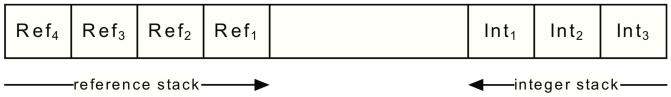

Figure 3.4: Darjeeling split operand stack, source: [13]

16-bit architecture Besides statically linking classes, the infuser also makes several changes to the bytecode format. References on sensor nodes are usually 16-bit, and 8-bit and 16-bit integer variables are commonly used in sensor node code to save memory. Storing all data in 32-bit slots as the JVM does would lead to significant overhead, as shown in Figure 3.3. Therefore, Darjeeling's stack and variable slots are 16-bit wide, using two slots for 32-bit ints, similar to how the normal JVM uses two 32-bit slots for 64-bit longs. Darjeeling's bytecode also introduces 16-bit versions of many opcodes, for example `SADD` adds two 16-bit shorts, while `IADD` adds two 32-bit ints. The infuser analyses the type of expressions, and replaces the 32-bit instructions found in normal JVM bytecode with the 16-bit variants where possible.

Double-ended stack A second important modification is that Darjeeling splits the operand stack into separate reference and integer stacks, as shown in Figure 3.4. Each stack frame still allocates the same amount of space for its operand stack, but Darjeeling places references on one side, and integers on the other. At the expense of having to keep track of two operand stack pointers, this reduces the complexity of the garbage collector significantly. When the garbage collector runs, it needs to find the *root set* of all live objects, which includes objects with references on the stack but not stored elsewhere. Since the type of the values on the operand stack changes continuously, it would be hard to determine which values are references and which are integers in a single stack, but using Darjeeling's

split stack design, it can simply place all values on the reference stack in the root set. For the same reason, Darjeeling also groups the slots allocated for local, static and instance variables into an integer and reference group.

The necessary modifications to the bytecode are taken care of by the infuser. Generic stack instructions such as `pop` are replaced by distinct versions for the integer and reference stack: `ipop` and `apop`.

Limitations Darjeeling implements a significant subset of Java, but like all sensor node JVMs it needs to make some sacrifices to scale down to sensor node size. Specifically, it does not support the 64-bit `long` datatype or floating point variables. It does not support reflection, since the infusion process drops the necessary type information from its class files to significantly reduce their size. Darjeeling also does not support the `synchronized` modifier on static methods, but it can be easily simulated using synchronized blocks, which are supported.

3.5 Performance

While many different VMs have been published, only a few papers describing sensor node VMs contain detailed performance measurements, shown earlier in Table 1.1. Darjeeling [13] reports between 30x and 113x slowdown for 3 benchmarks in 16-bit and 32-bit variations, compared to the native C equivalent. Ellul [24] reports measurements on the TakaTuka VM [8] where the VM is 230x slower than native code, and consumes 150x as much energy. TinyVM [41] is between 14x and 72x slower than C, for a set of 9 benchmarks. DVM [9] has different versions of the same benchmark, where the fully interpreted version is 108x slower than the fully native version, while Kumar reports a slowdown of 555x for the same VM [49]. Finally, SensorScheme [27] is up to 105x slower. Since performance depends on many factors, it is hard to compare these numbers directly. But the general picture is clear: current interpreters are one to two orders of magnitude slower than native code.

Translating bytecode to native code to improve performance has been a common prac-

tice for many years. A wide body of work exists exploring various approaches, either offline, ahead-of-time or just-in-time. One common offline method is to first translate the Java code to C as an intermediate language, and take advantage of the high quality C compilers available [23, 66]. Courbot et al. describe a different approach, where code size is reduced by partly running the application before it is loaded onto the node, allowing them to eliminate code that is only needed during initialisation [20]. Although the initialised objects are translated to C structures that are compiled and linked into a single image, the bytecode is still interpreted. While in general we can produce higher quality code when compiling offline, doing so sacrifices the key advantages of using a VM.

Hsieh et al. describe an early ahead-of-time compiling desktop Java VM [42], focussing on translating the JVM's stack-based architecture to a register based one. In the Japaleño VM, Alpern et al. take an approach that holds somewhere between AOT and JIT compilation [1]. The VM compiles all code to native code before execution, but can choose from two different compilers to do so. A fast baseline compiler simply mimics the Java stack, but either before or during run time, a slower optimising compiler may be used to speed up critical methods.

Since JIT compilers work at run time, much effort has gone into making the compilation process as light weight as possible. For example Krall [48] reduced the compilation time in early JIT compilers using a more lightweight register allocation algorithm. While the goals are similar to ours, this approach requires three passes over the code and significantly more complex data structures than a sensor node could handle. More recently these efforts have included JIT compilers targeted specifically at embedded devices. Swift [104] is a light-weight JVM that improves performance by translating a register-based bytecode to native code. But while the Android devices targeted by Swift may be considered embedded devices, they are still quite powerful and the transformations Swift does are too complex for the ATmega class of devices. HotPathVM [29] has lower requirements, but at 150 KB for both code and data, this is still an order of magnitude above our target devices.

Given our extreme size constraints - ideally we only want to use in the order of 100 bytes of RAM to allow our approach to be useful on a broad range of devices, and leave

Table 3.1: Example of Ellul's AOT translation of `c=a+b;`, source: [24]

Bytecode	Stack before	Stack after	Native pseudo assembly code
ILOAD_0	...	...	LOAD r1, a
	...	..., value1	PUSH r1
ILOAD_1	..., value1	..., value1	LOAD r1, b
	..., value1	..., value1, value2	PUSH r1
IADD	..., value1, value2	..., value1	POP r1
	..., value1	...	POP r2
	...	...	ADD r1, r2
	...	..., result	PUSH r1
ISTORE_2	..., result	...	POP r1
	...	...	STORE c, r1

ample space for concurrent tasks running on the device - almost all AOT and JIT techniques found in literature require too much resources. Indeed, some authors suggest sensor nodes are too restricted to make AOT or JIT compilation feasible [7, 102].

3.6 AOT compilation for sensor nodes

On the desktop, VM performance has been studied extensively, but for sensor node VMs this aspect has been mostly ignored. To the best of our knowledge AOT compilation on a sensor node has only been done by Ellul and Martinez [25, 24], and this work builds on theirs.

Their approach is both simple and effective. To translate the JVM bytecode, each bytecode instruction is replaced by a fixed sequence of native instructions that implements it. This can be done in a single pass, as the bytecode is being received by the node, writing blocks of native code to flash memory instead of JVM bytecode. Like Darjeeling, they use a split stack architecture, with the CPU's native stack doubling as integer operand stack, while reference operands are still stored in the stack frame.

Table 3.1 shows how a simple statement is translated to native code. The blocks each JVM bytecode instruction translates to are fixed. This simple translation to native code removes the interpreter loop, which is by far the biggest source of overhead in interpreting VMs, but it is clear from the native pseudo code in Table 3.1 that this approach results in many unnecessary push and pop instructions. Since the JVM is a stack-based VM, each

Table 3.2: Ellul's peephole optimisations

Before Instructions	Cycles	Length	After Instructions	Cycles	Length
PUSH R13 POP R13	6	2		0	0
PUSH R13 POP R14	6	2	MOV R13, R14	1	1
MOV #0, R15	2	2	CLR R15	2	1
MOV R6,R5 MOV R5,0x0000(R4)	5	3	MOV R6,0x0000(R4)	4	2

MSP430 assembly, source: [24]

instruction first obtains its operands by popping them from the stack and pushes any result back onto it. As a result, over half the instructions are push or pop instructions.

3.6.1 Peephole optimisation

To reduce this overhead, Ellul proposes a simple peephole optimiser [24] which does 4 optimisation. For each an example is shown in Table 3.2. The first to are the most important for improving performance: if a push is immediately followed by a pop to the same register, both are removed since they have no net effect. If the source and destination registers differ, the two instructions are replaced by a move. Note that the assembly code shown here is for the MSP430 CPU used in Ellul's work.

3.6.2 Resulting performance

Ellul's approach improves performance considerably compared to interpreters, but using the standard *CoreMark* benchmark [88], it generates code that is still up to 9.1x slower than optimised native C.

It is important to further improve this for a number of reasons. First, even if an application is asleep for a large percentage of the time, it may at times want to measure something at high resolution, similar to how ADAE [15] responds to what it calls 'interesting events' by taking extra measurements as long as its energy budget permits. Any slowdown in the VM will affect the maximum rate at which such samples can be taken. Similarly, Table

1.4 shows that for the Amulet smart watch platform, a 9.1x slowdown in the application code means the CPU is not fast enough to finish the some of the application's tasks in time. At best this would result in slower response time or sampling rates, at worst in could result in incorrect behaviour.

Second, reduced performance means the CPU has to stay active for a longer time, resulting in increased cpu power consumption and reduced battery life. Looking at the calculations for lossless compression in Section 1.2.1, the ratio of the energy saved on radio transmission vs energy spent on compression was about 2.6:1. This result depends on many factors, for instance less than ideal network conditions may cause retransmissions, increasing the cost per bit sent and making compression more worthwhile. However in this particular case the slowdown incurred even after Ellul's optimisation would make compression a net loss. Any slowdown will push some situations past the break-even point, or reduces the benefits of compression in others.

There are scenarios that may not be able to tolerate the one to two orders of magnitude slowdown seen in interpreters, but where a slowdown of up to 9.1x for Ellul's AOT approach may be acceptable. However, there is a third reason to improve on it: it results in code which is on average 3.0 times larger than the native equivalent. This reduces the amount of code we can load onto a node, and given that flash memory is already restricted, this is a major sacrifice to make when adopting AOT compilation on sensor nodes.

3.7 Safety

With some exceptions [27], most current sensor node VMs do not discuss safety, but instead focus on the functionality provided and how this can be implemented on a tiny sensor node. This is unfortunate, because the ability to provide a safety execution environment is both desirable, and easier to implement using a VM than it is using native code.

Several non-VM systems exists to provide safety for sensor nodes. They fall into two distinct categories, as shown in Figure 3.5. If we trust the host, it can verify the code and add run-time safety checks where needed, before the code is sent to the node. However, to allow the node to receive code from untrusted sources, for example to support mobile

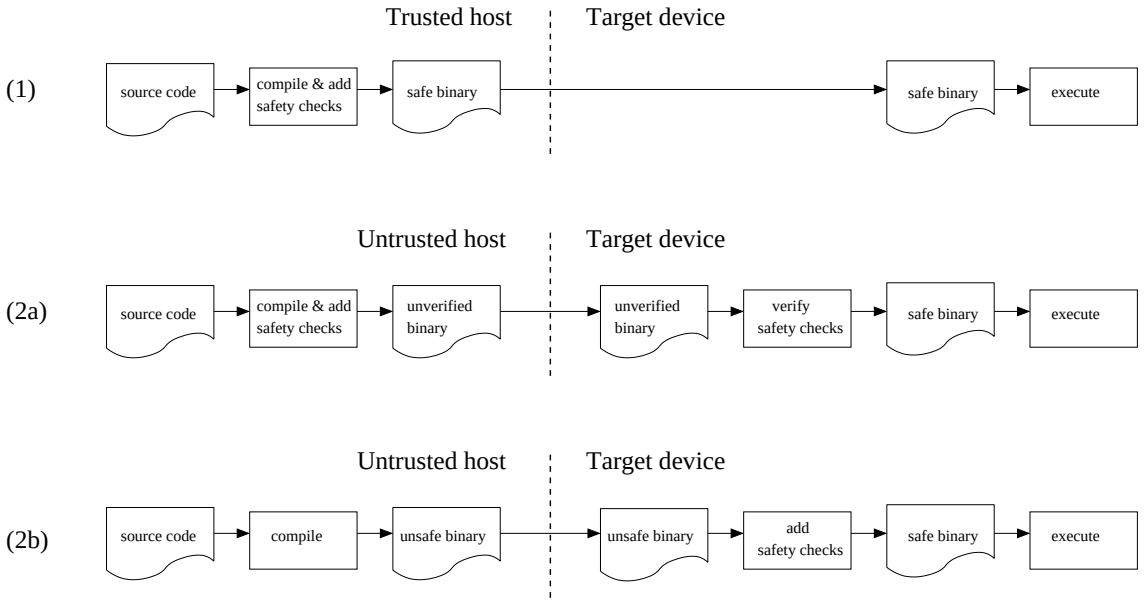

Figure 3.5: Three approaches to provide a safe execution environment

agents as in Agilla [28], it needs to guarantee safety independent of a host to guard against bugs and malicious attacks. The latter is obviously the stronger guarantee, but it also comes at a higher price.

3.7.1 Source code approaches

In the first category are systems that guarantee safety at the source code level. Virgil [89] is a language that is inherently safe and specifically designed for sensor nodes. The application is explicitly split into an initialisation and run-time phase, where objects are only allocated during the initialisation phase. The initialisation phase happens during compilation (to C code), which means all object and their locations are know at this point, allowing Virgil to ensure safety and optimise the code at compile time.

Safe TinyOS [19] on the other hand, works on annotated nesC TinyOS code. It uses the Deputy [18] source-to-source compiler to analyse the C source code and insert the necessary run-time checks were necessary. Because this happens on the host, before sending the code to the node, it can use the host's resources to do more complex analysis of the source code and eliminate checks where the it can determine a memory access to be safe at compile time, resulting in a much lower overhead

Similarly, applications for the Amulet [40] smart watch platform are written in a re-

stricted version of C that removes many of C's riskier features. Run-time checks are then added for operations where static analysis cannot guarantee them to be safe. Ongoing research in the Amulet project is on using hardware memory protection units found in some CPUs to reduce the cost of these restrictions and safety checks [39], but at the time of writing no results had been published.

All three approaches eventually result in standard C, which is then compiled and sent to the node. Therefore, these approaches may protect against accidental programming errors, but not against malicious code sent to a node.

3.7.2 Native code approaches

For desktop applications, Wahbe et al. described software fault isolation [95] techniques to isolate a piece of untrusted code, without the overhead of using processes and the CPU's memory protection. A typical example is a plugin that frequently needs to interact with an application. It should be isolated from the application so bugs in the plugin cannot bring down the whole application, but running it as a separate process would incur a high overhead.

Two basic methods are described to isolate such code from the main application: the native code can be rewritten at load time, inserting checks at all potentially unsafe writes, or it can be compiled to a predefined format with the appropriate checks already in place, after which the system loading the code only verifies it adheres to this standard at load time.

Since we do not have processes or CPU memory protection on a sensor node, Wahbe's approach provides an attractive alternative. Two systems exist that provide safety for sensor nodes using each of these approaches.

t-kernel [35] does more than providing memory safety. It raises the level of system abstraction for the developer by providing three features typically missing on sensor nodes: preemptive scheduling, virtual memory, and memory protection. It does this by extensive rewriting of the binary code at load time. While *t-kernel* is heavily optimised, the price for this is that programmes still run 50-200% slower, and code size increases by 500-750%.

The other approach is taken by Harbor [49], which consists of two components. On the desktop, a binary rewriter sandboxes an application by inserting run-time checks before it is sent to the node. The SOS operating system [37] is extended with a binary verifier to verify incoming binaries have the necessary checks in place. The correctness only depends on the correctness of this verifier. The increase in code size is more modest than for *t-kernel* at a 30-65% increase, but performance is 160-1230% slower. They also note more complex analysis of the binary code could reduce the number of necessary checks, but at the cost of significantly increasing the complexity of the verifier.

Finally, Weerasinghe and Coulson [98] proposed a system for module isolation similar to Harbor in the sense that code is compiled to a restricted, safe form of native code which is then verified by the node. Memory is allocated in fixed sized, aligned blocks, where a block size of 256 is suggested to simplify the verifier since the split between block address and offset falls nicely along byte boundaries. The authors aim to minimise run-time overhead and identify *t-kernel* code size overhead as a major drawback of this approach. Unfortunately their implementation was not yet finished when their paper was published and no further publications on this approach could be found.

Chapter 4

CapeVM

This chapter we introduce the global design of CapeVM, which is a modified version of Darjeeling [13]. It runs directly on the hardware, without any OS layer in between, and has complete control over the device. CapeVM modifies Darjeeling in three ways:

- AOT compilation: Darjeeling is originally an interpreter. To improve performance, this interpreter is replaced with an Ahead-of-Time compiler: instead of interpreting the bytecode, CapeVM translates it to native code at load time, before the application is started. While JIT compilation is possible on some devices [24], it depends on the ability to execute code from RAM, which many embedded CPUs, including the ATmega, cannot do.
- Bytecode format: To support the AOT compilation process and further improve performance, CapeVM modifies Darjeeling's bytecode format, and adds several new bytecode instructions.
- Safety: CapeVM adds a number of translation-time and run-time safety checks to provide a safe, sandboxed execution environment.

4.1 Goals

Working on resource-constrained devices means we have to make some compromises. Our main goal is to build an AOT compiling VM that generates safe code, performs well,

reduces the code size overhead seen in previous work, and fits as many IoT scenarios as possible. This includes scenarios like Amulet or WuKong, where multiple applications may be running on a single device. When new code is being loaded, the impact on other applications should be as small as possible.

WuKong, discussed in Section 3.2 is a good motivating example. Parts of WuKong applications may be written in Java. A node may be part of more than one application, and the WuKong master may dynamically decide to move an WuObject from one node to another. This means new Java code may have to be loaded onto a device, while parts of the same or another application are already running.

To support scenarios like this, the translation process should be very light weight. Specifically, it should use as little memory as possible, since this is a scarce resource and any memory used by the AOT compiler cannot be used for other concurrently running tasks. This means we cannot do any analysis on the bytecode that would require us to hold large data structures in memory.

Our goal is to limit memory use to around 100 bytes, which rules out most traditional AOT and JIT compiler techniques. It may be possible to achieve even better performance through more complex optimisations, but, as we will see, much can be achieved even with very limited resources.

The two metrics we compromise on are load time and code size. Compiling to native code takes longer than simply storing bytecode and starting the interpreter, but we feel this load-time delay will be acceptable in many cases, and will be quickly compensated for by improved run-time performance. Native code is already larger than JVM bytecode, and our safe, AOT compiled code is on average 82% larger than its optimised C equivalent. This is the price we pay for increased performance and safety, but the optimisations we propose do significantly reduce this code size overhead compared to previous work, thus reducing an important drawback of previous techniques.

We summarise our goals and constraints below:

- Improve performance to within half of native optimised code.
- Provide the option to run applications in a safe, sandboxed environment.

- Keep memory consumption to around 100 bytes to allow concurrent tasks to keep running and maintain their state as new code is loaded onto the device.
- Limit code size overhead so the approach is feasible on mid-range sensor node devices: at least 64 KB of programme memory for 8-bit Atmel CPUs.

The traditional argument to justify the performance overhead of sensor node VMs is that applications can tolerate a certain slowdown since the CPU will be in sleep mode most of the time, and that energy spent on the radio outweighs the energy spent on the CPU. Whether this is true depends both on the magnitude of the slowdown, and on the application.

The one to two orders of magnitude slowdown seen in interpreting VMs will be a problem for many applications, as illustrated by the three examples in Chapter 1. Ellul's work [24] on AOT compilation significantly reduced this performance overhead to a level that will be acceptable for many more applications. However, the remaining performance overhead of up to 9x is still a problem for the more demanding configurations of the Amulet smart watch, may make LEC compression a net loss, and Ellul's approach introduces a 3x code size overhead, which means less code can be loaded onto a device.

The optimisations presented in this dissertation reduce the slowdown to 1.7x, and the size of generated code by 40%. This further expands the range of scenarios for which a VM is a viable option.

Additional optimisations may improve performance even more, but on resource-constrained devices where CPU cycles, energy, RAM, and flash memory are all restricted, there is always a trade-off. At a slowdown of 1.7x, the tradition argument that this is acceptable because the CPU will still be in sleep mode for most of the time and other factors are more important in energy consumption, will hold for many more applications compared to a 9x or 100x slowdown. Adding more complex optimisations may rule out the use of a VM in scenarios that cannot afford the increase in VM size or the additional state that needs to be kept in RAM, while the improvement in performance is unlikely to enable many applications and has only limited impact on the total energy consumption. Thus, we believe CapeVM to be at a sensible point in this trade-off and Section 7.4.1 will show how

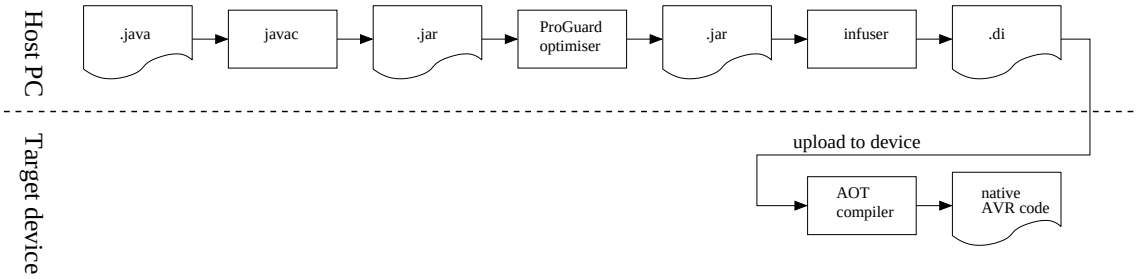

Figure 4.1: Java source to native AVR code compilation

selectively including optimisations gives it some flexibility to adjust to specific devices and scenarios.

4.2 Compilation process

The complete process from Java source to native code on the node is shown in Figure 4.1. Like all sensor node JVMs, CapeVM uses a modified JVM bytecode. Java source code is first compiled to normal Java classes. We then use ProGuard [36] to optimise the bytecode, but these optimisations, while useful, are limited to very basic steps such as dead code removal, overlapping local variable slots where possible, etc.

The optimised Java classes are then transformed into CapeVM's own bytecode format, called an *infusion*. For details of this transformation we refer to the Darjeeling paper [13] and the extensions to Darjeeling's bytecode described in Chapter 5. Here it is sufficient to note that no knowledge of the target platform is used in this transformation, so the result is still platform independent. This infusion is then sent to the node, where the AOT compiler translates it to native AVR code at load time.

When the node receives a large programme, it should not have to keep multiple messages in memory since this will consume too much memory. CapeVM's AOT compiler allows the bytecode to be translated to native code in a single pass, one instruction at a time. Only some small, fixed-size data structures are kept in memory during the process. A second pass over the generated code then fills in addresses left blank by branch instructions, since the target addresses of forward branches are not known until the target instruction is generated.

Table 4.1: Translation of `do{A>>>=1;} while(A>B);`

Bytecode	AOT compiler	AVR	cycles
0: BRTARGET(0)	*« record current address »*		
1: SLOAD_0	emit_LDD(R1,Y+0)	LDD R1,Y+0	4
	emit_PUSH(R1)	PUSH R1	4
2: SCONST_1	emit_LDI(R1,1)	LDI R1,1	2
	emit_PUSH(R1)	MOV R2,R1	1
3: SUSHR	emit_POP(R2)		
	emit_POP(R1)	POP R1	4
	emit_RJMP(+2)	RJMP +2	2
	emit_LSR(R1)	LSR R1	2
	emit_DEC(R2)	DEC R2	2
	emit_BRPL(-2)	BRPL -2	3
	emit_PUSH(R1)		
4: SSTORE_0	emit_POP(R1)		
	emit_STD(Y+0,R1)	STD Y+0,R1	4
5: SLOAD_0	emit_LDD(R1,Y+0)	LDD R1,Y+0	4
	emit_PUSH(R1)	PUSH R1	4
6: SLOAD_1	emit_LDD(R1,Y+2)	LDD R1,Y+2	4
	emit_PUSH(R1)		
7: IF_SCMPGT(BT:0)	emit_POP(R1)		
	emit_POP(R2)	POP R2	4
	emit_CP(R1,R2)	CP R1,R2	2
	emit_branchtag(GT,0)	BRGT 0:	2 (taken), or 1 (not taken)

Each message, which can be as small as a single bytecode instruction, can be freed immediately after processing. Since messages do need to be processed in the correct order, the actual transmission protocol may still decide to keep more messages in memory to reduce the need for retransmissions in the case of out of order delivery. But the translation process does not require it to do so, and a protocol that values memory usage over retransmissions cost can simply discard out of order messages and request retransmissions when necessary.

4.3 Translating bytecode to native code

The baseline AOT compiler used in CapeVM is an implementation of Ellul's approach, as described in his thesis [24], adapted for the Atmel ATmega128 CPU, while Ellul's work uses the Texas Instruments MSP430.

In this unoptimised version of the translator, each bytecode instruction the VM receives is simply replaced with a fixed, equivalent sequence of native instructions. The native stack is used to mimic the VM's operand stack. An example of this is shown in Table 4.1.

The first column shows a fragment of bytecode which does a shift right of 16-bit vari-

Table 4.2: CapeVM's peephole optimisations

Before Instructions	Cycles	Bytes	After Instructions	Cycles	Bytes
PUSH Rx POP Rx	4	4		0	0
PUSH Rx POP Ry	4	4	MOV Ry, Rx	1	2
ST X+, Rx LD -X, Rx	4	4		0	0
ST X+, Rx LD -X, Ry	4	4	MOV Ry, Rx	1	2
MOV Ry, Rx MOV Ry+1, Rx+1	2	4	MOVW Ry, Rx	1	2

The X register is used as the reference stack pointer.

able `A`, and repeats this while `A` is greater than `B`. While this may not be a very useful operation, it is the smallest example that will allow us to illustrate our code generation optimisations in the following chapter. The second column shows the code the AOT compiler will execute for each bytecode instruction. Together, the first and second column match the case labels and body of a big switch statement in the compiler. The third column shows the resulting AVR native code, which is currently almost a 1-on-1 mapping, with the exception of the branch instruction and some optimisations by a simple peephole optimiser, both described below.

The example has been slightly simplified for readability. Since the AVR is an 8-bit CPU, in the real code many instructions are duplicated for the high and low bytes. The cycle count is based on the actual number of generated instructions, and for a single iteration.

4.3.1 Peephole optimisation

Since the baseline should be as close as possible to Ellul's implementation, a similar set of peephole optimisations were implemented. However, differences between the ATmega and MSP430 instruction sets means they are not completely identical. The complete set of peephole optimisations in CapeVM is shown in Table 4.2.

A `push` directly followed by a `pop` are both either eliminated or replaced by a `mov`.

A push or pop may be either a real `push` or `pop` instruction for the integer stack, or implemented using a `st X+` or `ld -X` instruction for the reference stack. Both cases are optimised in the same way. We also similarly optimise blocks of pushes followed by blocks of pops, which are very common on the 8-bit ATmega.

When the push and pop instructions target different registers, this results in multiple `mov` instructions. Two `mov` instructions with consecutive registers can be further optimised using the `movw` instruction to copy a register pair to another register pair in a single cycle.

4.3.2 Branches

Forward branches pose a problem for this direct translation approach since the target address is not yet known. A second problem is that on the ATmega, a branch may take 1 to 3 words, depending on the distance to the target, so it is also not known how much space should be reserved for a branch.

To solve this, the infuser modifies the bytecode by inserting a new instruction, `BRTARGET`, in front of any instruction that is the target of a branch. The branch instructions themselves are modified to target the id of a `BRTARGET`, which are implicitly numbered, instead of a bytecode offset. When the VM encounters a `BRTARGET` during translation, no code is emitted, but the address where the next instruction will be emitted is recorded in a separate part of flash memory. Branch instruction initially emit a temporary 3-word 'branch tag', containing the branch target id and the branch condition. After code generation is finished and all target addresses are known, the VM scans the generated code a second time, and replaces each branch tag with the real branch instruction.

There is still the matter of the different sizes a branch may take. The VM could simply add `NOP` instructions to smaller branches to keep the size of each branch at 3 words, but this causes both a code size penalty and a performance penalty on small, non-taken branches. Instead, the VM does another scan of the code, before replacing the branch tags, to update the branch target addresses by compensating for cases where a smaller branch will be used. This second scan adds about 500 bytes to the VM, but improves performance, especially

on benchmarks where branches are common.

This is an example of something we often see: an optimisation may take a few hundred bytes to implement, but its usefulness may depend on the characteristics of the code being run. In this work we usually decided to implement these optimisations, since in many cases, including this one, they also result in smaller generated code.

4.3.3 Safety checks

CapeVM provides a safe execution environment by adding a number of safety checks, described in Chapter 6. An advantage of using a VM to provide safety is that bytecode is more structured than native code. This simplifies the necessary checks, and allows us to perform most of them at translation time.

For each bytecode instruction, a set of checks is defined to guarantee the generated code cannot violate the safety guarantees by writing or branching to an illegal address, under- or overflowing the stack, etc. Like the translation process, these checks are performed one instruction at a time, and only require minimal state to be maintained as a method is being translated, specifically two bytes to keep track of the operand stack.

Most of these checks are performed at translation time, and the VM will reject the code if any of them fail. For the remaining cases, run-time checks are included in the generated code that allow the VM to terminate any faulty application.

4.3.4 Bytecode modification

We made several modifications to the infuser and introduced new bytecode instructions to support our AOT compiler and improve performance. These changes will be introduced in more detail in the following chapters, but for completeness we also list them here:

- The `BRTARGET` opcode marks targets of branch instructions. All branch instructions are modified to target a `BRTARGET` id instead of a bytecode offset.

- The `MARKLOOP` opcode marks inner loops and the variables they use.

- `PUTFIELD_A_FIXED` and `GETFIELD_A_FIXED` are used to access an object's reference fields when the offset is known at compile time. The offset is always known at compile time for integer fields.
- The `SIMUL` opcode does 16x16-bit to 32-bit multiplication.
- New `_CONST` versions of the (variable) bit shift opcodes support shifting by a constant number of bits.
- The `INVOKELIGHT` opcode supports an optimised 'lightweight' way of calling methods.
- The `GETCONSTARRAY` opcodes support reading from arrays of constant data stored in the constant pool.
- Array access opcodes use 16-bit instead of 32-bit indexes.

4.3.5 Separation of integers and references

An important feature of Darjeeling, which we have maintained in CapeVM, is its separation of integers and references. When the garbage collector runs, it needs to mark the *root set*: the set of all live, directly reachable objects. This set is then iteratively expanded to include all indirectly reachable objects. For example in Figure 2.1, only the two objects on the left of the heap are in the root set, two others are also reachable but not in the root set, while the fifth is not reachable and will be freed by the garbage collector.

To mark the root set, the garbage collector needs to determine which local variables, static variables, object fields, and values on the operand stack are references. To do this efficiently, references and integers are separated throughout the VM: the operand stack, instance variables on the heap, class static variables, and local variables in a method's stack frame are all split in a block for integers and one for references, as shown in Figure 4.2.

In CapeVM's AOT compiler we use the native stack for the VM's integer operand stack, so the integer operand stack is no longer in the method's stack frame, but the ref-

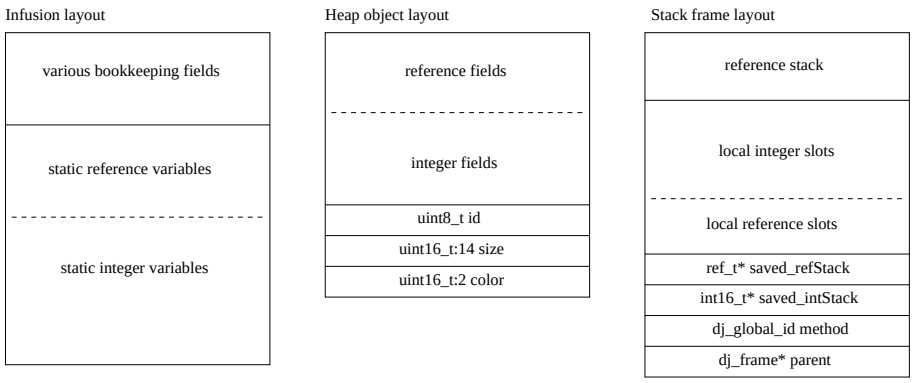

Figure 4.2: Infusion, object and stack frame layout

erence stack is. This uses less memory than having the integer stack in the stack frame, since we need to reserve space for the maximum stack depth in the frame, which is often much lower for the reference stack than for the integer stack. We use the AVR's X register as an extra stack pointer for the reference stack.

4.4 Limitations

Since CapeVM is based on Darjeeling, we share its limitations, most notably a lack of floating point support and reflection. In addition, we do not support threads or exceptions because after compilation to native code, we lose the interpreter loop as a convenient place to switch between threads or unwind the stack to jump to an exception handler. Threads and exceptions have been implemented in Ellul's AOT compiler [24], proving it is possible to add support for both, but we feel the added complexity in an environment where code space is at a premium makes other, more lightweight models for concurrency and error handling more appropriate. Dropping support for threads also allows us to allocate the VM's stack frames on the native stack, which considerably improves performance compared to Darjeeling's approach of allocating stack frames as a linked list on the heap.

We will discuss the cost of using our VM more in more detail in Chapter 7, and alternatives to some traditional JVM features in Chapter 8.

4.5 Target platforms

CapeVM was implemented for the ATmega128 CPU [65]. The AVR family of CPUs is widely used in low power embedded systems and sensor nodes. However, our approach does not depend on any AVR specific properties and we expect similar results for many other CPUs in this class. The main requirements are the ability to reprogramme its own programme memory, and the availability of a sufficient number of registers.

The ATmega128 has 32 8-bit registers. We expect the Cortex M0 [4], with 13 32-bit general purpose registers, or the MSP430 [87], with 12 16-bit registers, and used by Ellul and Martinez [25], to both be good matches as well. In Section 7.9 we will examine the expected impact of the number of registers and of using a 16-bit or 32-bit architecture on the resulting performance.

Chapter 5

Performance and code size optimisations

Having introduced our baseline AOT compiler, in this chapter we propose several optimisations to improve its performance and reduce the size of the generated code. We will first analyse the difference sources of overhead, and discuss how to reduce each of them.

5.1 Sources of overhead

The performance and code size of the baseline approach still suffers from a large overhead compared to optimised native C. To improve on this, it is important to identify the causes of this overhead. The main sources of overhead we found are:

- Lack of optimisations in the Java compiler
- AOT translation overhead
  - Push/pop overhead
  - Load/store overhead
  - JVM instruction set limitations
- Method call overhead

We will briefly discuss each source below, before introducing optimisations to reduce it.

5.1.1 Lack of optimisation in `javac`

A first source of overhead comes from the fact that the standard `javac` compiler does almost no optimisations. Since the JVM is an abstract machine, there is no clear performance model to optimise for. Run-time performance depends greatly on the target platform and the VM implementation running the bytecode, which are unknown when compiling Java source code to JVM bytecode. The `javac` compiler simply compiles the code 'as is'. For example, the loop

```
while (a < b*c) { a*=2; }
```

will evaluate '`b*c`' on each iteration, while it is clear that the result will be the same every time.

In most environments this is not a problem because the bytecode is usually compiled to native code before execution, and using knowledge of the target platform and the run-time behaviour, a desktop JIT compiler can make much better decisions than `javac`. However, since our AOT compiler simply replaces each instruction with a native equivalent, this leads to significant overhead in our VM.

We do use the ProGuard optimiser [36], but this only does very basic optimisations such as method inlining and dead code removal, and does not cover cases such as the example above.

5.1.2 AOT translation overhead

Assuming we have high quality JVM bytecode, a second source of overhead comes from the way the bytecode is translated to native code. We distinguish three main types of translation overhead, where the first two are a direct result of the JVM's stack-based architecture.

Type 1: Pushing and popping values

The compilation process initially results in a large number of push and pop instructions. In our simple example in Table 4.1, the peephole optimiser was able to eliminate some, but two push/pop pairs remain. For more complex expressions this type of overhead is higher, since more values will be on the stack at the same time. This means more corresponding push and pop instructions will not be consecutive, and the baseline peephole optimiser cannot eliminate these cases.

Type 2: Loading and storing values

The second type is also due to the JVM's stack-based architecture. Each operation consumes its operands from the stack, but in many cases the same value is needed again soon after. Because the value is no longer on the stack, this results in another load from memory.

In Table 4.1, it is clear that the `LDD` instruction at label 5 is unnecessary since the value is already in R1.

Type 3: JVM instruction set limitations

A third source of overhead due to the baseline AOT compilation process comes from optimisations that are done in native code, but are not possible in JVM bytecode, at least not in our resource-constrained environment.

The JVM instruction set is very simple, which makes it easy to implement, but this also means some things cannot be expressed as efficiently in bytecode as in native code. Given enough processing power, compilers can do the complex transformations necessary to make the compiled JVM code run almost as fast as native C, but a sensor node does not have such resources and must simply execute the instructions as they are.

The code in Table 4.1 does a shift right by one bit. In the JVM instruction set there is no way to express a single bit shift directly. Instead the constant 1 is loaded onto the stack, followed by the generic bit shift instruction. Compare this to addition, where the JVM bytecode does have a special INC instruction to add a constant value to a local variable.

A second example is arrays of constant data. Since the JVM has no concept of constant

data, any such data is implemented as a normal array, which has two disadvantages: it will use up precious RAM, and it will be initialised using normal JVM instructions, taking up much more code space than the constant data itself.

5.1.3 Method call overhead

The final source of overhead comes from method calls. In the JVM, each method has a stack frame (or 'activation frame') which the language specification describes as

> "containing the target reference (if any) and the argument values (if any), as well as enough space for the local variables and stack for the method to be invoked and any other bookkeeping information that may be required by the implementation (stack pointer, programme counter, reference to previous activation frame, and the like)" [33]

CapeVM's stack frame layout was shown in Figure 4.2. Initialising this complete structure is significantly more work than a native C function call has to do, which may not need a stack frame at all if all the work can be done in registers. Below we list the steps CapeVM goes through to invoke a Java method:

1. Flush the stack cache so parameters are in memory and clear value tags (see sections 5.3.2 and 5.3.3).
2. Save the integer and reference stack pointers (SP and X).
3. Call the VM's `callMethod` function, which will:
   (a) allocate memory for the callee's frame
   (b) initialise the callee's frame
   (c) pass parameters: pop them off the caller's stack and copy them into the callee's locals
   (d) activate the callee's frame: set the VM's active frame pointer to the callee
   (e) lookup the address of the AOT compiled code
   (f) do the actual `CALL`, which will return any return value in registers R22 and higher

Table 5.1: List of optimisations per overhead source

	Source of overhead	Optimisation
Section 5.2	Lack of optimisations in `javac`	• Manual optimisation of Java source code
Section 5.3	AOT translation overhead	
	Push/pop overhead	• Improved peephole optimiser
		• Stack caching
	Load/store overhead	• Popped value caching
		• Mark loops
	JVM instruction set limitations	• Constant bit shift optimisation
		• `GET/PUTFIELD_A_FIXED` instructions
		• `SIMUL` instruction
		• 16-bit array indexes
		• Support for constant arrays
Section 5.4	Method call overhead	• Lightweight methods

(g) reactivate the old frame: set the VM's active frame pointer back to the caller

(h) return to the caller's AOT compiled code the return value (if any) in R22 and higher

4. Restore stack pointer and X register.

5. Push the return value onto the stack (using stack caching this will be free).

Even after considerable effort optimising this process, this requires roughly 540 cycles for the simplest case: a call to a static method without any parameters or return value. For a virtual method the cost is higher because we need to look up the right implementation. While we may be able to save some more cycles with an even more rigorous refactoring, it is clear that the number of steps involved will always take considerably more time than a native function call.

5.1.4 Optimisations

Having identified these sources of overhead, we will use the next three sections to describe the set of optimisations we use to address them. Table 5.1 lists each optimisation, and the source of overhead it aims to reduce. The following sections will discuss each optimisation in detail.

5.2 Manually optimising the Java source code

As shown in Figure 4.1, our current implementation uses three steps to translate Java source code to CapeVM bytecode: the standard Java compiler, the ProGuard optimiser, and the modified Darjeeling infuser. None of these do any complex optimisations.

In a future version, these three steps should be merged into an 'optimising infuser' which uses all the normal, well-known optimisation techniques to produce better quality bytecode, but at the moment we do not have the resources to build such an optimising infuser.

Since our goal is to find out what level of performance is possible on a sensor node, the Java source were manually optimised to get better quality JVM bytecode from `javac`. While these changes are not an automatic optimisation we developed, we find it imporant to mention them explicitly and analyse their impact, since many developers may expect these optimisations to happen automatically and without this step it would be impossible to reproduce our results.

The most common optimisations we performed are:

- Store the result of expressions calculated in a loop in a temporary variable, if it is known the result will be the same for each iteration.
- Since array and object field access is relatively expensive and not cached by the mark loop optimisation discussed in Section 5.3.4, prefer to minimise array and object access by using a temporary local variable, if the value may be used again soon.
- Manually inline `#define`s and functions inlined by `avr-gcc`.
- Prefer to use 16-bit variables for array indexes where possible.
- Use bit shifts for multiplications by a power of two.

These are optimisations that the infuser cannot do automatically, but a developer who's aware of the performance of the VM could easily do manually. We will examine the effect of some other optimisations on the *CoreMark* benchmark in Section 7.2.

Temporary variables The first two optimisations generate extra store instructions, which means they may not always be beneficial if the value is never used again. But a value often only needs to be reused only once for it to be faster to store in a local variable than to calculate it twice or do two array accesses. If we use the mark loops optimisation discussed in Section 5.3.4, in many cases the variable may be stored in registers, making accessing them either very cheap, or completely free.

Manual inlining ProGuard automatically inlines methods that are only called from a single location, or small methods called from multiple locations. But when it does, it simply replaces the `INVOKE` instruction with the callee's body, prepended with `STORE` instructions to pop the parameters off the stack and initialise the callee's local variables. Manual inlining often results in better code, because it may not be necessary to store the parameters if they are only used once. Again, it is easy to imagine that an optimising infuser should be able to come to the same result automatically.

We therefore manually inline all small methods that were either a `#define` in the C version of our benchmarks, or a function that was inlined by `avr-gcc`.

Platform independence Assuming an optimising infuser does raise the question how platform independent the resulting code is. If the infuser has more specific knowledge about the target platform, it can produce better code for that platform, but, while it should still run anywhere, this may not be as efficient on other platforms.

However, the optimisations described here are only based on very conservative assumptions, and will work well for most devices in this class.

Example Listing 3 shows an example of these manual optimisations, applied to the *bubble sort* benchmark. To have a fair comparison, we applied exactly the same optimisations to the C versions of our benchmarks, but here this had little or no effect on the performance.

```java
// ORIGINAL
public static void bsort(int[] numbers)
{
  short NUMBERS=(short)numbers.length;
  for (short i=0; i<NUMBERS; i++)
  {
    for (short j=0; j<NUMBERS-i-1; j++)
    {
      if (numbers[j]>numbers[j+1])
      {
        int temp = numbers[j];
        numbers[j] = numbers[j+1];
        numbers[j+1] = temp;
      }
    }
  }
}
```

```java
// MANUALLY OPTIMISED
public static void bsort(int[] numbers)
{
  short NUMBERS=(short)numbers.length;
  for (short i=0; i<NUMBERS; i++)
  {
    short x=(short)(NUMBERS-i-1);
    short j_plus_1 = 1;
    for (short j=0; j<x; j++)
    {
      int val_at_j = numbers[j];
      int val_at_j_plus_1 = numbers[j_plus_1];
      if (val_at_j>val_at_j_plus_1)
      {
        numbers[j] = val_at_j_plus_1;
        numbers[j_plus_1] = val_at_j;
      }
      j_plus_1++;
    }
  }
}
```

Listing 3: Optimisation of the bubble sort benchmark

5.3 AOT translation overhead

Now that we have good quality bytecode to work with, we can start addressing the overhead incurred during the AOT compilation process.

5.3.1 Improving the peephole optimiser

Our first optimisation is a small but effective extension to the simple peephole optimiser. Instead of optimising only consecutive push/pop pairs, we can optimise any pair of push/pop instructions if the following holds for the instructions in between:

```
    PUSH Rs
    ..
    ..       instructions in between: - contain the same number of push and pop instr.
    ..                                - contain no branches
    ..                                - do not use register Rd
    ..
    POP  Rd
```

In this case the pair can be eliminated if Rs == Rd, otherwise it is replaced by a '`mov Rd, Rs`'. Two push/pop pairs remained in our earlier example Table 4.1. For the pair in instructions 5 and 7, the value is popped into register R2. Since instruction 6 does not use register R2, we can safely replace this pair with a direct move. In contrast, the pair in instructions 1 and 3 cannot be optimised since the value is popped into register R1, which is also used by instruction 2. The result is shown in Table 5.2

Table 5.2: Improved peephole optimiser

Bytecode	AOT compiler	AVR	cycles
0: BRTARGET(0)	*« record current address »*		
1: SLOAD_0	emit_LDD(R1,Y+0)	LDD R1,Y+0	4
	emit_PUSH(R1)	PUSH R1	4
2: SCONST_1	emit_LDI(R1,1)	LDI R1,1	2
	emit_PUSH(R1)	MOV R2,R1	1
3: SUSHR	emit_POP(R2)		
	emit_POP(R1)	POP R1	4
	emit_RJMP(+2)	RJMP +2	2
	emit_LSR(R1)	LSR R1	2
	emit_DEC(R2)	DEC R2	2
	emit_BRPL(-2)	BRPL -2	3
	emit_PUSH(R1)		
4: SSTORE_0	emit_POP(R1)		
	emit_STD(Y+0,R1)	STD Y+0,R1	4
5: SLOAD_0	emit_LDD(R1,Y+0)	LDD R1,Y+0	4
	emit_PUSH(R1)	MOV R2, R1	1
6: SLOAD_1	emit_LDD(R1,Y+2)	LDD R1,Y+2	4
	emit_PUSH(R1)		
7: IF_SCMPGT(BT:0)	emit_POP(R1)		
	emit_POP(R2)		
	emit_CP(R1,R2)	CP R1,R2	2
	emit_branchtag(GT,0)	BRGT 0:	2 (taken), or 1 (not taken)

5.3.2 Simple stack caching

The improved peephole optimiser can remove part of the type 1 overhead, but still many cases remain where it cannot eliminate the push/pop instructions. We use a form of stack caching [26] to eliminate most of the remaining push/pop overhead. Stack caching is not a new technique. It was originally proposed for Forth interpreters in 1995. But the trade-offs involved are very different depending on the scenario it is applied in, and it turns out to be exceptionally well suited for a sensor node AOT compiler:

First, the VM in the original paper is an interpreter, which means the stack cache has to be very lightweight, or the overhead from managing it at run time will outweigh the time saved by reducing memory accesses. Since we only need to keep track of the cache state at translation time, this restriction does not apply for an AOT compiler and we can afford to spend more time managing it. Second, the simplicity of the approach means it requires very little memory: only 11 bytes of RAM and less than 1 KB of code more than the peephole optimiser.

The basic idea of stack caching is to keep the top elements of the stack in registers instead of main memory. We add a cache state to our VM to keep track of which registers are holding stack elements. For example, if the top two elements are kept in registers, an

Table 5.3: Simple stack caching

Bytecode	AOT compiler	AVR	cycles	cache state R1	R2	R3	R4
0: BRTARGET(0)	*« record current address »*						
1: SLOAD_0	operand_1 = sc_getfreereg()			*			
	emit_LDD(operand_1,Y+0)	LDD R1,Y+0	4	*			
	sc_push(operand_1)			Int1			
2: SCONST_1	operand_1 = sc_getfreereg()			Int1	*		
	emit_LDI(operand_1,1)	LDI R2,1	2	Int1	*		
	sc_push(operand_1)			Int2	Int1		
3: SUSHR	operand_1 = sc_pop()			Int1	*		
	operand_2 = sc_pop()			*	*		
	emit_JMP(+2)	JMP +2	2	*	*		
	emit_LSR(operand_2)	LSR R1	2	*	*		
	emit_DEC(operand_1)	DEC R2	1	*	*		
	emit_BRPL(-2)	BRPL -2	1	*	*		
	sc_push(operand_2)			Int1	*		
4: SSTORE_0	operand_1 = sc_pop()			*			
	emit_STD(Y+0,operand_1)	STD Y+0,R1	4	*			
5: SLOAD_0	operand_1 = sc_getfreereg()			*			
	emit_LDD(operand_1,Y+0)	LDD R1,Y+0	4	*			
	sc_push(operand_1)			Int1			
6: SLOAD_1	operand_1 = sc_getfreereg()			Int1	*		
	emit_LDD(operand_1,Y+2)	LDD R2,Y+2	4	Int1	*		
	sc_push(operand_1)			Int2	Int1		
7: IF_SCMPGT(BT:0)	operand_1 = sc_pop()			Int1	*		
	operand_2 = sc_pop()			*	*		
	emit_CP(operand_1, operand_2);	CP R2,R1	2	*	*		
	emit_branchtag(GT, 0);	BRGT 0:	2	*	*		

`ADD` instruction does not need to access main memory, but can simply add these registers, and update the cache state. Values are only spilled to memory when no more free registers are available.

In the baseline AOT approach, each bytecode instruction maps to a fixed sequence of native instructions that always use the same registers. Using stack caching, the registers are controlled by a stack cache manager that provides three functions:

- `sc_getfree`: Instructions such as load instructions will need a free register to load the value into, which will later be pushed onto the stack. If all registers are in use, `sc_getfree` spills the register that's lowest on the stack to memory by emitting a `PUSH`, and then returns that register. Thus, the top of the stack is kept in registers, while lower elements may be spilled to memory.

- `sc_pop`: Pops the top element off the stack and tells the code generator in which register it can be found. If stack elements have previously been spilled to main memory and no elements are left in registers, `sc_pop` will emit a real `POP` instruction to get the value back from memory.

- `sc_push`: Updates the cache state so the passed register is now at the top of the stack. This should be a register that was previously returned by `sc_getfree`, or `sc_pop`.

Using stack caching, code generation is split between the code generator, which emits the instructions that do the actual work, and the cache manager which manages the registers and may emit code to spill stack elements to memory, or to retrieve them again. But as long as enough registers are available, it will only manipulate the cache state.

In Table 5.3 the same example is translated, but this time using stack caching. The `emit_PUSH` and `emit_POP` instructions have been replaced by calls to the cache manager, and instructions that load something onto the stack start by asking the cache manager for a free register. The state of the stack cache is shown in the four columns added to the right. Currently it only tracks whether a register is on the stack or not. ”Int1” marks the top element, followed by ”Int2”, etc. This example does not use the reference stack, but it is cached in the same way as the integer stack. A ”*” marks a register that is marked as being used by the current instruction, but not currently on the stack. The next two optimisations will extend the cache state further.

The example only shows four registers, but the ATmega128 has 32 8-bit registers. Since CapeVM uses a 16-bit stack, they are managed as pairs. 10 registers are reserved, for example as a scratch register or to store a pointer to local variables, leaving 11 pairs available for stack caching.

Branches Branch targets may be reached from multiple locations. We know the cache state if it was reached from the previous instruction, but not if it was reached through a branch. To ensure the cache state is the same on both paths, the whole stack is flushed to memory whenever we encounter either a branch or a `BRTARGET` instruction.

This may seem bad for performance, but fortunately the stack is empty at almost all branches in the code generated by `javac`. A common exception is the ternary `? :` operator, which may cause a conditional branch with elements on the stack, but in most cases flushing at branches and branch targets does not result in any extra overhead.

Table 5.4: Popped value caching

Bytecode	AOT compiler	AVR	cycles	cache state R1	R2	R3	R4
0: BRTARGET(0)	*« record current address »*						
1: SLOAD_0	operand_1 = sc_getfreereg()			*			
	emit_LDD(operand_1,Y+0)	LDD R1,Y+0	4	*			
	sc_push(operand_1)			Int1LS0			
2: SCONST_1	operand_1 = sc_getfreereg()			Int1LS0	*		
	emit_LDI(operand_1,1)	LDI R2,1	2	Int1LS0	*		
	sc_push(operand_1)			Int2LS0	Int1CS1		
3: SUSHR	operand_1 = sc_pop_destructive()			Int1LS0	*		
	operand_2 = sc_pop_destructive()			*	*		
	emit_JMP(+2)	JMP +2	2	*	*		
	emit_LSR(operand_2)	LSR R1	2	*	*		
	emit_DEC(operand_1)	DEC R2	1	*	*		
	emit_BRPL(-2)	BRPL -2	1	*	*		
	sc_push(operand_2)			*	Int1		
4: SSTORE_0	operand_1 = sc_pop_tostore()				*LS0		
	emit_STD(Y+0,operand_1)	STD Y+0,R2	4		*LS0		
5: SLOAD_0	*« skip codegen, update cache state »*				Int1LS0		
6: SLOAD_1	operand_1 = sc_getfreereg()			*	Int1LS0		
	emit_LDD(operand_1,Y+2)	LDD R1,Y+2	4	*	Int1LS0		
	sc_push(operand_1)			Int1LS1	Int2LS0		
7: IF_SCMPGT 0:	operand_1 = sc_pop_nondestructive()			*LS1	Int1LS0		
	operand_2 = sc_pop_nondestructive()			*LS1	*LS0		
	emit_CP(operand_1, operand_2);	CP R1,R2	2	*LS1	*LS0		
	emit_branchtag(GT, 0);	BRGT 0:	2	*LS1	*LS0		

5.3.3 Popped value caching

Stack caching can eliminate most of the push/pop overhead, even when the stack depth increases. We now turn our attention to reducing the overhead resulting from load and store instructions.

A *value tag* is added to each register's cache state to keep track of what value is currently held in the register, after it is popped from the stack. Some bytecode instructions have a value tag associated with them to indicate which value or variable they load, store, or modify. Each tag consist of a tuple (type, datatype, number). For example, the instructions `ILOAD_0` and `ISTORE_0`, which load and store the local integer variable with id 0, both have tag LI0, short for (local, int, 0). `SCONST_1` has tag CS1, or (constant, short, 1), etc. These tags are encoded as a 16-bit value.

The cache manager is extended with a `sc_can_skip` function. This function will examine the type of each instruction, its value tag, and the cache state. If it finds that we are loading a value that is already present in a register, it updates the cache state to put that register on the stack, and returns true to tell the main loop to skip code generation for this instruction.

Table 5.4 shows popped value caching applied to our example. At first, the stack is empty. When `sc_push` is called, it detects the current instruction's value tag, and marks the fact that R1 now contains LS0. In `SUSHR_CONST`, the `sc_pop` has been changed to `sc_pop_destructive`. This tells the cache manager that the value in the register will be destroyed, so the value tag has to be cleared again since R1 will no longer contain LS0. The `SSTORE_0` instruction now calls `sc_pop_tostore` instead of `sc_pop`, to inform the cache manager it will store this value in the variable identified by `SSTORE_0`'s value tag. This means the register once again contains LS0. If any other register was marked as containing LS0, the cache manager would clear that tag, since it is no longer accurate after we update the variable.

In bytecode instruction 5, we need to load LS0 again, but now the cache state shows that LS0 is already in R1. This means it does not need to load it from memory, but the cache manager can just update the cache state so that R1 is pushed onto the stack. At run time this `SLOAD_0` will have no cost at all.

There are a few more details to get right. For example if a value is loaded that's already on the stack, a move is emitted to copy it. When `sc_getfree` is called, it will try to return a register without a value tag. If none are available, the least recently used register is returned. This is done to maximise the chance we can reuse a value later, since recently used values are more likely to be used again.

Branches As we do not know the state of the registers if an instruction is reached through a branch, we have to clear all value tags when we pass a `BRTARGET` instruction, meaning that any new loads will have to come from memory. At branches we can keep the value tags, because if the branch is not taken, the state of the registers in the next instruction is known.

5.3.4 Mark loops

Popped value caching reduces the type 2 overhead significantly, but the fact that we have to clear the value tags at branch targets means that a large part of this overhead still remains.

Table 5.5: Mark loops

Bytecode	AOT compiler	AVR	cycles	cache state R1	R2	R3	R4
0: MARKLOOP(0,1)	*« emit markloop prologue: »*	LDD R1,Y+0	4	LS0PIN			
	« LS0 and LS1 are live »	LDD R2,Y+2	4	LS0PIN	LS1PIN		
1: BRTARGET(0)	*« record current address »*			LS0PIN	LS1PIN		
2: SLOAD_0	*« skip codegen, update cache state »*			Int1LS0PIN	LS1PIN		
3: SCONST_1	operand_1 = sc_getfreereg()			Int1LS0PIN	LS1PIN	*	
	emit_LDI(operand_1,1)	LDI R3,1	2	Int1LS0PIN	LS1PIN	*	
	sc_push(operand_1)			Int2LS0PIN	LS1PIN	Int1CS1	
4: SUSHR	operand_1 = sc_pop_destructive()			Int1LS0PIN	LS1PIN	*	
	operand_2 = sc_pop_destructive()	MOV R4,R1	1	LS0PIN	LS1PIN	*	*
	emit_JMP(+2)	JMP +2	2	LS0PIN	LS1PIN	*	*
	emit_LSR(operand_2)	LSR R4	2	LS0PIN	LS1PIN	*	*
	emit_DEC(operand_1)	DEC R3	1	LS0PIN	LS1PIN	*	*
	emit_BRPL(-2)	BRPL -2	1	LS0PIN	LS1PIN	*	*
	sc_push(operand_2)			LS0PIN	LS1PIN	*	Int1
5: SSTORE_0	*« emit MOV, update cache state »*	MOV R1,R4	1	LS0PIN	LS1PIN		
6: SLOAD_0	*« skip codegen, update cache state »*			Int1LS0PIN	LS1PIN		
7: SLOAD_1	*« skip codegen, update cache state »*			Int2LS0PIN	Int1LS1PIN		
8: IF_SCMPGT(BT:0)	operand_1 = sc_pop_nondestructive()			Int1LS0PIN	LS1PIN		
	operand_2 = sc_pop_nondestructive()			LS0PIN	LS1PIN		
	emit_CP(operand_1, operand_2);	CP R2,R1	2	LS0PIN	LS1PIN		
	emit_branchtag(GT, 0);	BRGT 1:	2	LS0PIN	LS1PIN		
9: MARKLOOP(end)	*« emit markloop epilogue: LS0 is live »*	STD Y+0,R1	4	LS0	LS1		

This is particularly true for loops, since each iteration often uses the same variables, but the branch to start the next iteration clears those values from the stack cache. This is addressed by the next optimisation.

Again, we modify the infuser to add a new instruction to the bytecode: `MARKLOOP`. This instruction is used to mark the beginning and end of each innermost loop. `MARKLOOP` has a larger payload than most bytecode instructions: it contains a list of value tags that will appear in the loop and how often each tag appears, sorted in descending order.

When we encounter the `MARKLOOP` instruction, the VM may decide to reserve a number of registers and pin the most frequently used local variables to them. If it does, code is generated to prefetch these variables from memory and store them in registers. While in the loop, loading or storing these pinned variables does not require memory access, but only a manipulation of the cache state, and possibly a simple move between registers. However, these registers will no longer be available for normal stack caching. Since 4 register pairs need to be reserved for code generation, at most 7 of the 11 available pairs can be used by mark loops.

Because the only way to enter and leave the loop is through the `MARKLOOP` instructions, the values can remain pinned for the whole duration of the block, regardless of the

branches made inside. This lets us eliminate more load instructions, and also replace store instructions by a much cheaper move to the pinned register. `INC` instructions, which increment a local variable, operate directly on the pinned register, saving both a load and a store. All these cases are handled in `sc_can_skip`, bypassing the normal code generation. A small change to `sc_pop_destructive` is also necessary. If the register that is about to be popped is pinned, it cannot be used directly since this would corrupt the value of the pinned local variable. Instead, first a move to a free, non-pinned register, is emitted.

In Table 5.5 the first instruction is now `MARKLOOP`, which tells the compiler local short variables 0 and 1 will be used. The compiler decides to pin them both to registers 1 and 2. The `MARKLOOP` instruction also tells the VM whether or not the variables are live, which they are at this point, so the two necessary loads are generated. This is reflected in the cache state. No elements are on the stack yet, but register 1 is pinned to LS0, and register 2 to LS1.

Next, LS0 is loaded. Since it is pinned to register 1, no code is generated, but the cache state is updated to reflect LS0 is now on top of the stack. After loading the constant 1, `SUSHR` pops both operands destructively. We cannot simply return register 1 since that would corrupt the value of variable LS0, so the second `sc_pop_destructive` emits a move to a free register and returns that register instead. Since LS0 is pinned, `SSTORE_0` can also be skipped, but `sc_can_skip` does need to emit a move back to the pinned register.

The next two loads are straightforward and can be skipped, and in the branch the registers are popped non-destructively, so the pinned registers can be used directly.

Finally, the loop ends with another `MARKLOOP`, telling the compiler only local 0 is live at this point. This means LS0 in register 1 needs to be stored back to memory, but LS1 can be skipped since it is no longer needed.

5.3.5 Instruction set modifications

Next, we introduce five optimisations that target the type 3 overhead: cases where limitations in the JVM instruction set means some operations cannot be expressed as efficiently

as in native code. This type of overhead is the most difficult to address because many of the transformations a desktop VM can do to avoid it take more resources than we can afford on a tiny device. Also, this type of overhead covers many different cases, and optimisations that help in a specific case may not be general enough to justify spending additional resources on it.

Still, there are a few things we can do by modifying the instruction set, that come at little cost to the VM and can make a significant difference.

Darjeeling's original instruction set is already quite different from the normal JVM instruction set. The most important change is the introduction of 16-bit operations. The JVM is internally a 32-bit machine, meaning **`short`**, **`byte`**, and **`char`** are internally stored as 32-bit integers. On a sensor device where memory is the most scarce resource, we often want to use shorter data types. To support this, Darjeeling internally stores values in 16-bit slots, and introduces 16-bit versions of all integer operations. For example, to multiply two shorts and store the result in a short, the 32-bit `IMUL` instruction is replaced by the 16-bit `SMUL` instruction. These transformations are all done by the infuser (see Figure 4.1).

However, the changes made by Darjeeling are primarily aimed at reducing memory consumption, not at improving performance. The infuser was extended to make several other changes. The `BRTARGET` and `MARKLOOP` instructions have already been discussed, and the `INVOKELIGHT` instruction is the topic of the next section. In addition to these, the following five other modifications were made to Darjeeling's instruction set:

Constant bit shifts

Most benchmarks described in Section 7 do bit shifts by a constant number of bits. These appear not only in computation intensive benchmarks, but also as optimised multiplications or divisions by a power of 2, which are common in many programmes.

In JVM bytecode the shift operators take two operands from the stack: the value to shift, and the number of bits to shift by. While this is generic, it is not efficient for constant shifts: the constant first needs to be pushed onto the stack, and the bit shift is implemented

Table 5.6: Constant bit shift optimisation

Bytecode	AOT compiler	AVR	cycles	cache state R1	R2	R3	R4
0: MARKLOOP(0,1)	*« emit markloop prologue: »*	LDD R1,Y+0	4	LS0PIN			
	« LS0 and LS1 are live »	LDD R2,Y+2	4	LS0PIN	LS1PIN		
1: BRTARGET(0)	*« record current address »*			LS0PIN	LS1PIN		
2: SLOAD_0	*« skip codegen, just update cache state »*			Int1LS0PIN	LS1PIN		
3: SUSHR_CONST(1)	*« operand_1 = sc_pop_destructive() »*	MOV R3,R1	1	LS0PIN	LS1PIN	*	
	« emit_LSR(operand_1) »	LSR R3	2	LS0PIN	LS1PIN	*	
	« sc_push(operand_1) »			LS0PIN	LS1PIN	Int1	
4: SSTORE_0	*« emit MOV, update cache state »*	MOV R1,R3	1	LS0PIN	LS1PIN		
5: SLOAD_0	*« skip codegen, just update cache state »*			Int1LS0PIN	LS1PIN		
6: SLOAD_1	*« skip codegen, just update cache state »*			Int2LS0PIN	Int1LS1PIN		
7: IF_SCMPGT(BT:0)	operand_1 = sc_pop_nondestructive()			Int1LS0PIN	LS1PIN		
	operand_2 = sc_pop_nondestructive()			LS0PIN	LS1PIN		
	emit_CP(operand_1, operand_2);	CP R2,R1	2	LS0PIN	LS1PIN		
	emit_branchtag(GT, 0);	BRGT 1:	2	LS0PIN	LS1PIN		
8: MARKLOOP(end)	*« emit markloop epilogue: LS0 is live »*	STD Y+0,R1	4	LS0	LS1		

as a simple loop which shifts one bit at a time. If the number of bits to shift by is already known, much more efficient code can be generated.

Note that this is different from other arithmetic operations with a constant operand. For operations such as addition, the translation process results in loading the constant and performing the addition, which is similar to what `avr-gcc` generates in most cases. An addition takes just as long when the operand is taken from the stack, as when it is a constant.

The mismatch is in the fact that while the JVM instruction set is more general, with both operands being variable for both bit shifts and other arithmetic operations, the native instruction set can only shift by a single bit. This means that to shift by a number of bits that is unknown until run time, a loop must be generated to shift one bit at a time, which is much slower than the code that can be generated for a shift by a constant number of bits.

We optimise these cases by adding `_CONST` versions of the bit shift instructions `ISHL`, `ISHR`, `IUSHR`, `SSHL`, `SSHR`, and `SUSHR`. We add a simple scan to the infuser to find constant loads that are immediately followed by a bit shift. For these cases the constant load is removed, and the bit shift instruction, for example `ISHL`, is replaced by `ISHL_CONST`, which has a one byte constant operand in the bytecode, containing the number of bits to shift by. On the VM side, implementing these six `_CONST` versions of the bit shift opcodes adds 444 bytes to the VM, but it improves performance, sometimes very significantly, for all but three of the benchmarks.

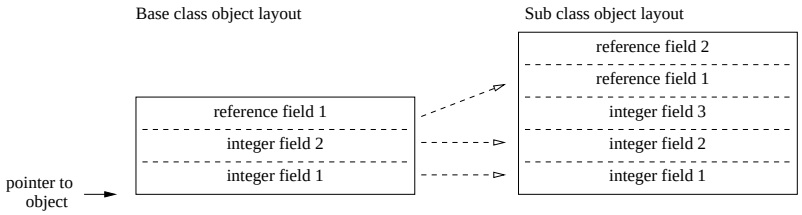

Figure 5.1: Base class and sub class layout

Surprisingly, after this was implemented, one benchmark performed better than native C. We found that `avr-gcc` does not optimise constant shifts in all cases. Since the goal is to examine how close a sensor node VM can come to native performance, it would be unfair to include an optimisation that is not found in the native compiler, but could easily be added. We implemented a version that is close to what `avr-gcc` does, but never better. The VM only considers cases that are optimised by `avr-gcc`. For these, first whole byte moves are emitted if the number of bits to shift by is 8 or more, followed by single bit shifts for the remainder.

The result when applied to our example is shown in Table 5.6, where the `SCONST_1` and `SUSHR` instructions have been replaced by a single `SUSHR_CONST` instruction. The total cost is now 20 cycles, but 12 of these are spent before and after the loop, while each iteration now only takes 8 cycles, a significant improvement from the 48 cycles spent in the original version in Table 4.1.

`GET/PUTFIELD_A_FIXED` reference field access

The `GETFIELD_*` and `PUTFIELD_*` instructions are used to access fields in objects. Because references and integer slots are split, the offset from the object pointer is known at compile time only for integer fields, but not for reference fields. As shown in Figure 5.1, integer fields will be at the same offset, regardless of whether an object is of the compile-time type, or a subclass. References fields may shift up in subclasses, so `GETFIELD_A` and `PUTFIELD_A` must examine the object's type at run time and calculate the offset accordingly, adding significant overhead.

This overhead can be avoided if we can be sure of the offset at compile time, which is the case if the compile-time type is marked **`final`**. In this case the infuser will replace

the `GETFIELD_A` or `PUTFIELD_A` opcode with a `_FIXED` version so the VM knows it is safe to determine the offset at AOT translation time. Conveniently, one of the optimisations ProGuard does, is to mark any class that is not subclassed as **`final`**, so most of this is automatic.

Alternative solutions An alternative we considered is to let go of the split for references and integers for object fields and mix them, so the offsets for reference fields would also be known at compile time. To allow the garbage collector to find the reference fields we could either extend the class descriptors with a bit map indicating the type of each slot, or let the garbage collector scan all classes in the inheritance line of an object.

We chose our solution because it is easier to implement and adds only a few bytes to the VM size, while the garbage collector is already one of the most complex components of the VM. Also, we found that almost all classes in our benchmarks could be marked **`final`**. Either solution would work, and the alternative could be considered as a more general solution.

Evaluation The impact of this optimisation is significant, but we decided not to include it in our evaluation since the overhead is the result of implementation choices in Darjeeling, which was optimised for size rather than performance. This means the overhead is a result of a Darjeeling specific choice, rather than a direct result of the AOT techniques or the JVM's design. Therefore, all results reported in this paper are with this optimisation already turned on.

Since the split architecture has many advantages in terms of complexity and VM size, we still feel it is important to mention this as an example of the kind of trade-offs faced when optimising for performance.

`SIMUL` 16-bit to 32-bit multiplication

While Darjeeling already introduced 16-bit arithmetic operations, it does not cover the case of multiplying two 16-bit shorts, and storing the result in a 32-bit integer. In this case the infuser would emit `S2I` instructions to convert the operands to two 32-bit integers, and

then use the normal `IMUL` instruction for full 32-bit multiplication. On an 8-bit device, this is significantly more expensive than 16x16 to 32-bit multiplication.

We added a new opcode, `SIMUL`, for this case, which the infuser will emit if it can determine the operands are 16-bit, but the result is used as a 32-bit integer.

We could add more instructions, for example, `SIADD` for 16-bit to 32-bit addition, `BSMUL` for 8-bit to 16-bit multiplication, etc. But there is a trade-off between the added complexity of an optimisation and the performance improvement it yields, and for these cases this is much smaller than for `SIMUL`.

16-bit array indexes

The normal JVM array access instructions (`IASTORE`, `IALOAD`, etc) use a 32-bit index. On a sensor node with only a few KB of memory, we will never have arrays that require such large indexes, so we modified the array access instructions to use a 16-bit index instead.

This complements one of the manual optimisations discussed in Section 5.2. Using short values as index variables makes operations on the index variable cheaper, while changing the operand of the array access instructions reduces the amount of work the array access instruction needs to do and the number of registers it requires.

In CapeVM's bytecode, the array access instructions are named `GETARRAY` and `PUTARRAY` to be in line with `GET/PUTFIELD` and `GET/PUTSTATIC`.

Support for constant arrays

Finally, 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, but on a sensor node 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 sensor

node applications. For example, the FFT benchmark contains an array of precalculated sine wave values. When we implement this as a **`final`** Java array, the compiler emits a static class initialiser that creates a normal Java array object, and then uses the array access instructions to initialise each element individually, as shown in Listing 4. The **`final`** keyword only affects the reference to the `Sinewave` array, but not the array itself.

```
1     private final static byte Sinewave[] = new byte[] {
2         0, 3, 6, 9, 12, 15, 18, 21,
3         ...
4         -24, -21, -18, -15, -12, -9, -6, -3,
5     };
```

```
1     sspush(256); newarray;                        // create the array
2     adup; sconst_0;    sconst_0;   bastore;       // set index 0
3     adup; sconst_1;    sconst_3;   bastore;       // set index 1
4     adup; sconst_2;    bspush(6);  bastore;       // set index 2
5     ...
6     adup; bspush(255); bspush(-3); bastore;       // set index 255
```

Listing 4: Array of constant data from the 8-bit FFT benchmark, and the resulting bytecode without the constant array optimisation

There are two problems with this: (i) the array will occupy scarce RAM; and (ii) initialising array elements using bytecode instructions requires 4 instructions per element, resulting in 1663 bytes of bytecode to initialise a 256 byte array, which expands even further after AOT compilation.

To solve this we introduce four new `GETCONSTARRAY` instructions to read from constant arrays of ints, shorts, chars or bytes. The normal `GETARRAY` instructions take two operands from the stack: the reference to the array, and the index of the element to load. The `GETCONSTARRAY` instructions only read the index from the stack, and have the id of an array in the constant pool encoded as a single byte operand in the bytecode instruction.

The infuser is modified to place constant arrays in the constant pool. Since the current infuser works on the `javac` output, it requires each constant array to be placed in a wrapper class with the `@ConstArray` annotation, which simplifies processing them considerably.

When the infuser loads a class and finds the `@ConstArray` annotation, it parses the `<clinit>` class initialiser to extract the data for the constant array. The class initialiser is then removed, and the data for the constant array is added to the constant pool.

When the infuser processes a method's bytecode, it analyses the operand stack. If a `GETSTATIC_A` instruction loads a reference to an array in a `@ConstArray` class onto the stack, the entry used for stack analysis is marked. This allows the infuser to the find the corresponding `GETARRAY` instruction that consumes this reference from the stack. The infuser then replaces it with a `GETCONSTARRAY` instruction, which carries the constant pool id of the array as a bytecode operand, and removes the original `GETSTATIC_A`.

This means the array no longer takes up any RAM, and only uses the amount of flash required to hold the data and a four byte header. Another advantage of this approach is that the reference no longer needs to be loaded onto the stack. This eliminates the `GETSTATIC_A` instruction, reduces the stack depth, and slightly simplifies the address calculation to find the correct offset since constant arrays do not have the small header used for normal arrays. This compensates for the fact that reading from flash is slightly more expensive at 3 cycles per byte, compared to 2 for reading from RAM.

Alternative solutions A disadvantage of the chosen approach is that the constant arrays cannot be used for anything except directly reading from them. Since no array object is created, we cannot assign a reference to it to any variable, for example to decide at run time whether to read from one array or another.

This would be possible if we keep the `GETSTATIC_A` instruction, but use it to load a special reference to the constant array onto the stack instead. However, this would remove the advantages of not having to load the reference and result in a more complex design.

Since all examples found in the benchmarks directly read from the constant arrays, we choose the simpler option, and note that such a run time decision could also be implemented using an `if` statement or a small helper function, albeit at a higher cost.

5.4 Method calls

Finally, we will look at the overhead caused by method calls. In native code, the smallest possible function call only has 8 cycles of overhead for a `CALL` and a `RET` instruction. More complicated functions may spend up to 76 cycles saving and restoring call-saved

registers. As seen in Section 5.1.3, in Java a considerable amount of state needs to be initialised. For the simplest method call this takes about 540 cycles, and this increases further for large methods with many parameters.

Methods in a programme typically form a spectrum from a limited number of large methods at the base of the call tree that take a long time to complete and are only called a few times, to small (near-)leaf methods that are fast and frequently called. Figure 5.2 shows this spectrum for the *CoreMark* benchmark.

For the slow methods at the base, the overhead of the method call is insignificant compared to the total execution time. However, as we get closer to the leaf methods, the number of calls increases, as does the impact on the overall performance.

At the very end of this spectrum are tiny helper functions that may be inlined, but this is only possible for very small methods, or methods called from a single place. In *CoreMark*'s case, `ee_isdigit` was small enough to inline. When larger methods are inlined, the trade-off is an increase in code size. There is a problem in the middle of the spectrum: methods that are too large to inline, but called often enough for the method call overhead to have a significant impact the overall performance.

5.4.1 Lightweight methods

For these cases we introduce a new type of method call: lightweight methods. These methods differ from normal methods in two ways:

- No stack frame is created for lightweight methods, but the caller's frame is used.
- Parameters are passed on the stack, rather than in local variables.

Lightweight methods give us third choice, in between a normal method call and method inlining. When calling a lightweight method, the method's AOT compiled code is called directly. This bypasses the VM completely, reusing the caller's stack frame, and leaving the parameters on the (caller's) stack. In effect, the lightweight methods behave similar to inlined code, but do not incur the code size overhead of duplicating large inlined methods. Because the method will be called from multiple locations which may have different cache

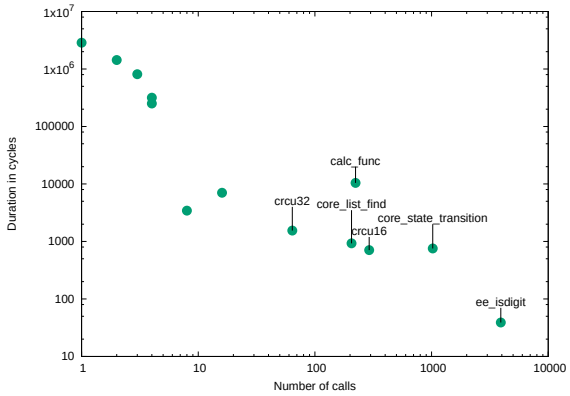

Figure 5.2: Number of CoreMark method calls vs. duration (logarithmic scales)

states, the stack cache must be flushed to memory before a call. This results in slightly more overhead than for inlined code, but much less than for a normal method call.

As an example, consider the simple `isOdd` method in Listing 5:

```
// JAVA

public static boolean
           isOdd (short a)
{
  return (a & (short)1)==1;
}
```

```
// NORMAL METHOD
//                (Stack)
SLOAD_0           (Int)
SCONST_1          (Int,Int)
SAND              (Int)
SRETURN           ()
```

```
// LIGHTWEIGHT METHOD
//                (Stack)
SCONST_1          (Int,Int)
SAND              (Int)
SRETURN           ()
```

Listing 5: Simple, stack-only lightweight method example

The normal implementation has a single local variable. It expects the parameter to be stored in the local variable and the stack to be empty when we enter the method. In contrast, the lightweight method does not have any local variables and expects the parameter to be on the stack at the start of the method.

We added a new instruction, `INVOKELIGHT`, to call lightweight methods. Listing 6 shows how `INVOKELIGHT` and `INVOKESTATIC` are translated to native code. Both first flush the stack cache to memory. After that, the lightweight method can directly call the implementation of `isOdd`, while the native version first saves the stack pointers, and then enters an expensive call into the VM to setup a stack frame for `isOdd`, which in turn will call the actual method.

```
// NORMAL INVOCATION
// INVOKESTATIC isOdd:
    push r25        // Flush the cache
    push r24
    call &preinvoke // Save X and SP
    ldi r22, 253    // Set parameters
    ldi r23, 2      //  for callMethod
    ldi r24, 21
    ldi r20, 64
    ldi r21, 42
    ldi r18, 13
    ldi r19, 0
    ldi r25, 2
    call &callMethod // Call to VM
    call &postinvoke // Restore X and SP
```

```
// LIGHTWEIGHT INVOCATION
// INVOKELIGHT isOdd:
    push r25      // Flush the cache
    push r24
    call &isOdd
```

Listing 6: Comparison of lightweight and normal method invocation

Local variables

The lightweight implementation of the `isOdd` example only needs to process the values that are on the stack, but this is only possible for the smallest methods. If a lightweight method has local variables, space is reserved for them in the caller's stack frame, equal to the maximum number of slots needed by all the lightweight methods it may call.

CapeVM uses the ATmega's Y register to point the start of a method's local variables. To call a lightweight method with local variables, the caller only needs to shift Y up to the region reserved for lightweight method variables before doing the `CALL`. The lightweight method can then access its locals as if it were a normal method. After the lightweight method returns, the caller lowers Y, so it points to the caller's own variables again.

Nested calls

A final extension is to allow for nested calls. While frequently called leaf methods benefit the most from lightweight methods, there are many cases where it is useful for one lightweight methods to call another. A good example from the *CoreMark* benchmark is the 32-bit `crcu32` function, which is implemented as two calls to `crcu16`. For the best performance, both methods should be lightweight.

So far we have not discussed how to handle the return address in a lightweight method. Our AOT compiler uses the native stack to store VM's integer stack value, which means the operands to a lightweight method will be on the native stack. But after a `CALL`, the return address is put on the stack, covering the method parameters.

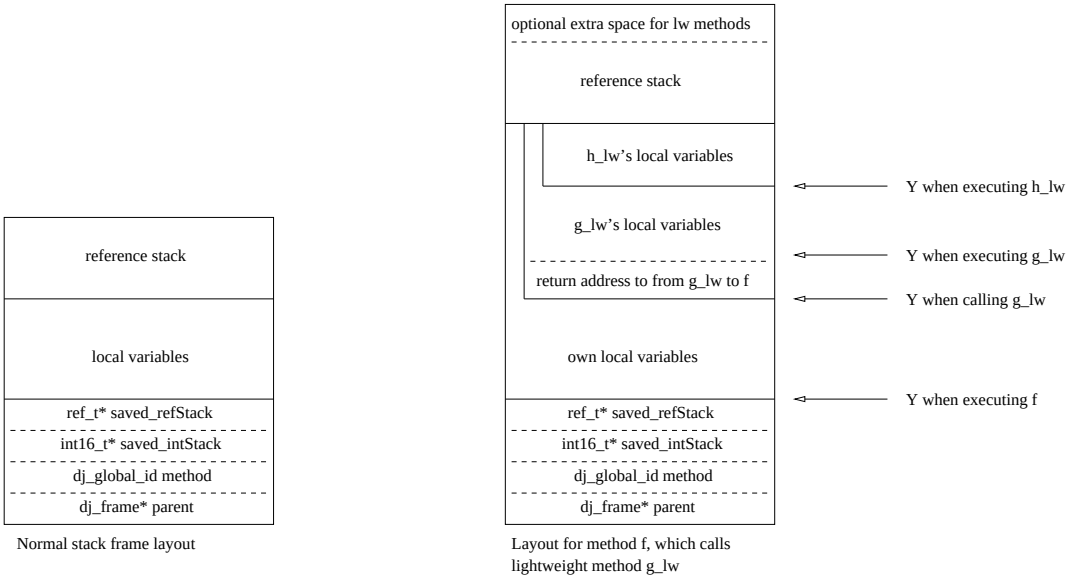

Figure 5.3: Stack frame layout for a normal method `f`, which calls lightweight method `g_lw`, which in turn calls lightweight method `h_lw`

For leaf methods, the lightweight method will first pop the return address into two fixed registers, and avoid using these register for stack caching. When the method returns, the return address is pushed back onto the stack just before the `RET` instruction.

For lightweight methods that will call another lightweight method, the return value is also popped from the stack, but instead of leaving it in the fixed register, where it would be overwritten by the nested call, we save it in the first local variable slot and increment Y to skip this slot. Since each lightweight method has its own block of locals, we can nest calls as deeply as needed.

This difference in method prologue and epilogue is the only difference in the way the VM generates code for a lightweight method, all bytecode instructions can then be translated the same way as for a normal method.

Stack frame layout

A normal method that invokes a possible string of lightweight methods, needs to save space for this in its stack frame. How much space it needs to reserve can be determined by the infuser at compile time, and this information is added to the method header used to create the stack frame.

An example is shown in Figure 5.3, which shows the stack frame for a normal method

`f`, which calls lightweight method `g_lw`, which in turn calls another lightweight method `h_lw`.

The stack frame for `f` contains space for its own locals, and for the locals of the lightweight method it calls: `g_lw`. In turn, `g_lw`'s locals contain space for `h_lw`'s locals, as well as a slot to store the return address back to `f`. Since `h_lw` does not call any other methods, it just keeps its return address in registers.

When a method calls a lightweight method with local variables, it will move the Y register to point to that method's locals. From Figure 5.3 it is clear it only needs to increment Y by the size of its own locals. For `f`, this will place the Y register at the beginning of `g_lw`'s locals. Since `g_lw` may call `h_lw`, `g_lw`'s prologue will first store the return address in the first local slot, moving Y forward in the process so that Y points to the first free slot.

Mark loop

Lightweight methods may use any register and do not save call-saved registers like normal methods. When a lightweight method is called inside a `MARKLOOP` block, it may corrupt some of the variables pinned to registers. In this case the caller saves those variables back to memory before calling the lightweight method and loads them again after the call returns. Since lightweight methods always come before their invocation in the infusion, the VM already knows which registers it uses, and will only save and restore pinned variables if there is a conflict. Because registers for `MARKLOOP` are allocated low to high, and for normal stack caching from high to low, in many cases the two may not collide.

Example call

An example of the most complex case for a lightweight call is shown in Listing 7, which shows how method `f` from Figure 5.3 would call `g_lw`, assuming `f` is in a `MARKLOOP` block at the time which pinned a variable R14:R15, and these registers are also used by `g_lw`.

In the translation of the `INVOKELIGHT` instruction, first the stack cache is flushed to

memory, then the value of the local variable at offset 22 is saved because it was pinned to R14:R15. Next, the Y register is incremented by 26 to skip the caller's own local variables and point Y to the start of the space reserved for lightweight method locals.

In the implementation of `g_lw`, the return address is popped off the stack into R18:R19. Since `g_lw` may call another lightweight method which will do the same, the return address is stored in the first local slot, incrementing Y in the process. After `g_lw`'s body, the return address is pushed back onto the stack before the final `ret` instruction.

After `g_lw` returns, the reverse process is used to return to the caller: the Y register is restored to point to the caller's locals, and the local variable at offset 22 is loaded back into the pinned registers R14:R15.

```
// LIGHTWEIGHT INVOCATION
INVOKELIGHT g_lw
    push r25        // Flush the cache
    push r24
    std Y+22, r14   // Save pinned value
    std Y+23, r15
    adiw Y, 26      // Move Y to g_lw's
    call &g_lw                  // locals
                                                    // IMPLEMENTATION OF g_lw
                                                        pop r18     // Pop the return address
                                                        pop r19
                                                        st Y+, r18  // Save in 1st local,
                                                        st Y+, r19  //  and increment Y

                                                          .. // g_lw's body

                                                        ld r19, -Y  // Load return address,
                                                        ld r18, -Y  //  and decrement Y
                                                        push r19    // Push return address
                                                        push r18    //  onto the stack
                                                        ret
    sbiw Y, 26      // Restore Y
    ldd r14, Y+22   // Reload pinned value
    ldd r15, Y+23
```

Listing 7: Full lightweight method call

5.4.2 Creating lightweight methods

We currently support two ways to create lightweight methods:

- Handwritten bytecode
- Converted Java methods

Handwritten bytecode

For the first option we declare the methods `native` in the Java source code, so the code calling it will compile as usual. We provide the infuser with a handwritten implementation in bytecode, which the infuser will simply add to the infusion, and then process it in the same way it processes a normal method, with one additional step:

For lightweight methods, the parameters will be on the stack at the start of the method, but the infuser expects to start with an empty stack. To allow the infuser to process them like other methods, we add a dummy `LW_PARAMETER` instruction for each parameter. This instruction is skipped when writing the binary infusion, but it tricks the infuser into thinking the parameters are being put on the stack.

Converted Java methods

This handwritten approach is useful for the smallest methods, and allows us to create bytecode that only uses the stack, which produces the most efficient code. But for more complex methods it quickly becomes very cumbersome to write the bytecode by hand.

As a second, slightly slower, but more convenient option, we provide a way to convert normal Java methods to lightweight methods by adding a `@Lightweight` annotation to it.

The infuser will scan all the methods in an infusion for this annotation. When it finds a method marked `@Lightweight`, the transformation to turn a normal method into a lightweight one is simple: we first add a dummy `LW_PARAMETER` instruction for each parameter, followed by `STORE` instructions to pop these parameters off the stack and store them in the right local variables. After this, we can use the normal body of the method and call it as a lightweight method.

Listing 8 shows the difference for the `isOdd` method. We can see this approach adds some overhead in the form of a `SSTORE_0` and a `SLOAD_0` instruction. However, using popped value caching, only the `SSTORE_0` will have a run-time cost. Another disadvantage of the converted method is that is has to use a local variable, which will slightly increase memory usage, but in return this approach gives us a very easy way to create

lightweight methods.

```
// JAVA
@Lightweight
public static boolean
          isOdd (short a)
{
  return (a & (short)1)==1;
}
```

```
// HANDWRITTEN
//              (Stack)
LW_PARAMETER    (Int)
SCONST_1        (Int,Int)
SAND            (Int)
SRETURN         ()
```

```
// CONVERTED JAVA
//              (Stack)
LW_PARAMETER    (Int)
SSTORE_0        ()
SLOAD_0         (Int)
SCONST_1        (Int,Int)
SAND            (Int)
SRETURN         ()
```

Listing 8: Comparison of hand written lightweight method and converted Java method

Replacing `INVOKE`s

The infuser does a few more transformations to the bytecode. Every method is scanned for `INVOKESTATIC` instructions that invoke a lightweight method. These are simply replaced by an `INVOKELIGHT` instruction, and the number of extra slots for the reference stack and local variables of the current method is increased if necessary. Finally, methods are sorted so a lightweight method will be defined before it is invoked, to make sure the VM can always generate the `CALL` directly.

5.4.3 Overhead comparison

We now compare the overhead for the various ways we can call a method in Table 5.7.

Manually inlining code yields the best performance, but at the cost of increasing code size if larger methods are inlined. ProGuard inlining is slightly more expensive because it always saves parameters in local variables.

Both lightweight methods options cause some overhead, although this is very little compared to a full method call. First, we need to flush the stack cache to memory to make sure the parameters are on the real stack. This this takes two `push` and eventually two corresponding `pop` instructions per word, costing 8 cycles per word. In addition, we need to clear the value tags from the stack cache, which means we may not be able to skip as many `LOAD` instructions after the lightweight call, but the effect of this is hard to quantify.

Next the cost of translating the `INVOKE` instruction varies depending on the situation. In the simplest case it is simply a `CALL` to the lightweight method, which together with

the corresponding `RET` costs 8 cycles. The worst case is 68 cycles when the lightweight method has local variables, uses all registers, and the caller used the maximum of 7 pairs to pin variables in a `MARKLOOP` block.

After calling the method, the method prologue for lightweight methods is very simple. It saves the return address and restores it in the epilogue, which takes 8 cycles if left in a register, or 16 if it needs to be stored in a local variable slot.

For small handwritten lightweight methods this is the only cost, but for larger ones created by converting a Java method, we add `STORE` instructions to copy the parameters from the stack into local variables, as shown in Listing 8. This is similar to the only overhead incurred by ProGuard's method inlining, and costs 4 cycles per word for the `STORE`, and possibly 4 more if the corresponding `LOAD` cannot be eliminated by popped value caching.

The total overhead for a lightweight method call scales nicely with the method's complexity. For the smallest methods, the minimum is only 16 cycles, plus 8 cycles per word for the parameters. For the most complex cases this may go up to 100 to 150 cycles. But these methods must be more complex and will have a longer run time, which limits the relative overhead.

The number of cycles in Table 5.7 is just a broad indication of the overhead. Some factors, such as the cost of clearing the value tags is hard to predict, and inlining may allow some optimisations that are not possible with a method call. In practice the actual cost in a number of specific cases we examined varies, but is in the range we predicted.

For a normal method call the cost is much higher, and less dependent on the complexity of the method that is called. The overhead from setting up the stack frame, and the more expensive translation of the `INVOKE` instruction (see Listing 5) are fixed, meaning a call will cost at least around 540 cycles, increasing to over 700 cycles for more complex methods taking many parameters.

5.4.4 Limitations and trade-offs

There are a few limitations to the use of lightweight methods:

Table 5.7: Approximate cycles of overhead caused by different ways of invoking a method

	Manual inlining	ProGuard inlining	Stack-only lightweight	Converted Java lightweight	Normal method call
flush the stack cache [1]			8 per word	8 per word	8 per word
`INVOKE`			8 to 68	8 to 68	~80
create stack frame					~450
method pro-/epilogue			8 or 16	8 or 16	10 to 71
store and load parameters		4 or 8 per word		4 or 8 per word	4 or 8 per word
total		*4 or 8 per word*	*16 to 84 + 8 per word*	*16 to 84 + 12 or 16 per word*	*~540 to ~601 + 12 or 16 per word*

No recursion Since we need to be able to determine how much space to reserve in the caller's stack frame for a lightweight method's reference stack and local variables, recursion is not supported, although lightweight calls can be nested. The limited amount of memory on a sensor node means recursion is a bad choice in most cases.

No garbage collection Lightweight methods reuse the caller's stack frame. This is a problem for the garbage collector, which works by inspecting each stack frame and finding the references on the stack and in local variables. If the garbage collector would be triggered while a lightweight method is being executed, it would not know where to find the lightweight method's references, since the stack frame only has information for the method that owns it. Thus, lightweight methods cannot perform any action that may trigger the garbage collector.

While it may be possible to relax this constraint with some effort, in most cases this is only a minor restriction. Lightweight methods are most useful for fast and frequently called methods, and operations that may trigger the garbage collector are usually expensive, so there is less to be gained from using a lightweight method in these situations.

Static only Lightweight virtual methods are not supported, but this is something that could be considered in future work.

Stack frame usage Finally, we should remember that a method calling a lightweight method always reserves space for it in its locals. This space is reserved, regardless of whether the method is currently executing or not, and the more nested lightweight calls

are made, the more space we need to reserve.

As an example, consider a method `f1` which may call a lightweight method with a large number of local variables, `big_lw`, but is currently calling normal method `f2`, which may also call `big_lw`. In this case space for `big_lw` will be reserved twice, both in `f1`'s and in `f2`'s frame.

Chapter 6

Safety

The second goal of this dissertation is to develop a VM that offers a 'safe' execution environment, and to compare the cost of doing so using a VM to existing native code approaches.

A safe execution environment is one that guarantees an application cannot harm the system it is running on or other applications running on the same system. Specifically, an application cannot:

1. Execute code it does not have permission for,
2. Write to memory outside the areas assigned to it, or
3. Retain control of the CPU indefinitely.

Given the first two, the last guarantee is easy to implement: the VM can use interrupts or set a timer to trap back to the VM to regain control when needed. As long as the other guarantees hold, the application will not be able to disable these without the VM's permission. To guard against programming errors as well as malicious attacks, we focus on the second type of approaches shown in Figure 3.5, where the node does not rely on a trusted host to guarantee safety, but can do so independent of the code it receives.

As discussed in Chapter 3, most generic sensor nodes VMs do not consider safety, with the exception of SensorScheme [27], but a number of native code systems have been proposed. This is unfortunate because using a VM has some distinct advantages compared to native code systems that guarantee safety.

Table 6.1: List of safety checks

	Translation-time checks
T-1	For each method header, the number of own local variable slots <= the number of total variable slots, the number of (int/ref) arguments <= the number of (int/ref) variables, static methods are not abstract.
T-2	The last instruction of each method is a `RETURN` or `GOTO`.
T-3	Branch instructions branch to an index < the number of `BRTARGET`s announced in the method header.
T-4	At the end of each method, we have seen the exact number of `BRTARGET` instructions announced in the method header.
T-5	The target for an `INVOKELIGHT` call is already translated, so the target address is known.
T-6	The target method header for an `INVOKESTATIC`/`INVOKESPECIAL` exists.
T-7	After popping the return value, the stack is empty.
T-8	At each `INVOKELIGHT` instruction, the max stack of the caller >= the current stack depth - the number of arguments to the callee + the max stack of the callee, for both integer and reference stacks.
T-9	Before each instruction, the stack depth >= the number of elements to be consumed by the instruction.
T-10	After each instruction, the stack depth <= the max stack depth announced in the header.
T-11	The stack is empty at branches and branch targets.
T-12	For each `INVOKELIGHT`, the total number of variable slots - the number of own variable slots for the caller >= the total variable slots for the callee.
T-13	The index of the local variable < the number of own variable slots for the current method.
T-14	The target infusion of a static variable exists.
T-15	The index of the static variable < the number of static variable slots for the target infusion.
	Run-time checks
R-1	The target implementation for an `INVOKEVIRTUAL`/`INVOKEINTERFACE` is found.
R-2	Whenever a new stack frame is allocated, the frame+max stack depth+some safety margin > the end of the heap.
R-3	The target implementation for an `INVOKEVIRTUAL`/`INVOKEINTERFACE` matches the stack effects used to verify the caller's stack at translation time.
R-4	The target address of an array element or object field is within the heap.
R-5	The headers of heap chunks form a consistent chain of chunks, ending at the byte indicated by a pointer to the first free heap byte.

Both the machine model and the instruction set of the virtual machine are more structured than a real CPU. This restricts what a bytecode instruction can do, which in turn allows the VM to verify safety for many instructions at translation time.

As an example, memory on the physical CPU is a flat address space, and the ATmega's store instructions can write to *any* address in memory, which means any write could potentially corrupt memory. In the VM, memory for local variables, static variables, and for objects and arrays are separated, and the VM has separate instructions to target each of them as an offset within their respective regions. Similarly, native code can branch to any address within flash memory, while the VM's branches target the id of a branch target within the current method.

While in CapeVM, the programme eventually also runs as native code on the physical CPU, the VM is in complete control of how this code is generated and the context in which it runs. As shown in the next sections, this make it easy to determine at translation time that stores to local and static variables will write to space reserved for a local or static variable, and that branches will branch to a legal instruction within the method.

Table 6.1 contains the list of checks done by CapeVM to guarantee safety. To show these are sufficient to satisfy the high level guarantees listed before, we first express them as concrete constraints specific to CapeVM:

- *Control flow safety*: after starting the application, the VM is always executing either
  - a translated bytecode instruction in the current method *from the start*, or
  - code in the VM itself, as a result of either a call to a VM function from a translated bytecode instruction, or returning from a method.
- *Memory safety*: any write to memory done by the application is to a legal location: either
  - memory reserved for the operand stack, or
  - a valid local or static variable slot, or
  - the area of the heap assigned to the application.

The control flow guarantees make sure code cannot jump to a point half-way a generated instruction to skip run-time checks, or to anywhere in the VM except through the proper entry points defined by the VM. As in normal Java, the bytecode instructions in CapeVM can only modify state within the virtual machine. There are no special instructions to access sensors and actuators as in for example Maté. Thus, access to external resources is assumed to happen through calls to natively implemented library functions, which returns control to VM and allows it to check whether the application has permission to do so.

Clearly, the VM will be in a legal state when a programme starts: the VM will jump to the begining of the first instruction and no writes will have occured yet. We will show

the checks in Table 6.1 are sufficient to ensure these two constraints, by examining how each bytecode instruction can affect control flow and write to memory, and list the checks necessary to ensure the VM is still in a legal state after executing it. Both guarantees depend on each other: memory safety is assumed when discussing control flow safety and vice versa.

Each method in CapeVM has a small *method header* defining properties such as the maximum stack size, number of local variables, return type, etc. The VM uses this header to create the stack frame, and to determine the effects of a call to the method on the caller's operand stack. Therefore, many of the checks are to ensure the implementation of the method follows the contract established in the method header. When the node receives new code, it first receives the headers for all methods, followed by their implementations, so the contracts for all methods are known when the bytecode is translated.

The first check is a basic sanity check on the data in the method headers (T-1). Since each parameter becomes a local variable, the number of local variable slots must be at least as high as the number of parameters, and the total number of slots must be at least as high as the method's own local variable slots, while the rest may be used for lightweight methods. Finally, a method cannot be marked static and abstract at the same time.

6.1 Control flow safety

We show control flow safety by considering all bytecode instructions, and showing they either flow into the start of a legal next instruction, or return control back to the VM. Regarding their effect on control flow, the bytecode instructions can be grouped into four categories, shown in Table 6.2. The state is correct at the start of the programme, since the VM will start it by jumping to the beginning of the first instruction in the main method. We will show the state will be correct after each following instruction by looking at these four categories.

Table 6.2: Instructions affecting control flow

Type	Effect on control flow
Branches	Jump to a location within the method
`INVOKE`	Call a method, either through the VM or directly
`RETURN`	Return to the address at the top of the stack
Others	Fall through to the next bytecode instruction

6.1.1 Simple instructions

Starting with the last category: most instructions such as math operations, loads and stores, are translated to a sequence of native instructions that will be executed top to bottom. In some cases this may call to a VM or `libc` function to perform some complex operation, temporarily handing control back to the VM, but these calls return to the current instruction once the operation is complete.

For this category, the generated code will flow naturally into the start of the next generated instruction. This is means the control flow constraint would be broken if there *is no* next instruction, which produces the second translation-time check, T-2: the last instruction in a method must be a `RETURN` or `GOTO` to prevent control from flowing out of the method body.

6.1.2 Branch instructions

In CapeVM bytecode, branches do not target an offset as in normal JVM bytecode, but the id of a branch target. These targets are marked with `BRTARGET` instructions, which do not emit any code, but cause the AOT compiler to collect the address in a temporary table during translation. Once the whole method is translated, this temporary table is used to patch the correct target address into the branch instructions.

Each method announces the number of branch targets that will be used in the method header. Non-taken branches flow into a correct next instruction because T-2 guarantees they are not the last instruction in a method. To ensure a taken branch will branch to the start of an instruction within the method, two checks are needed: the target ID of each branch must be lower than number of branch targets announced in the method header (T-3), and at the end of the method the number of `BRTARGET` instructions encountered must

be equal to the number announced in the header (T-4). The first guarantees each branch refers to an entry in the temporary table, while the second guarantees all entries are filled with a valid address.

6.1.3 Method invocation instructions

There are three kinds of method invocations.

For lightweight method calls, the implementation of the target method is required to come before any method invoking it, to ensure the address of the target is known at translation time and we can directly generate a `CALL` to the method. Ensuring this calls to a correct address is therefore trivial: for `INVOKELIGHT` instructions, the implementation of the target method must already have been generated (T-5).

For static calls (`INVOKESTATIC` and `INVOKESPECIAL`) the target method is known at translation time, but it may not have been translated yet if the implementation follows later in the infusion. For these instructions, a `CALL` to the VM's `callMethod` function is generated, passing the id of the target method as a parameter. At run time this id will be used as an index in the method table to find its implementation. Because the VM receives all the method headers before translating their implementations, it can check at translation time that the method id is known (T-6). Since the VM will not start an application before all methods are translated, this guarantees that `callMethod` will find the target at run time.

Finally, for `INVOKEVIRTUAL` and `INVOKEINTERFACE`, the target is not known at translation time, since this depends on the object the method is invoked on. Darjeeling does not use dispatch tables to call virtual methods. Instead, each infusion contains a flattened list of methods that are scanned to find the right implementation for the current object. Therefore a run-time check is needed to verify the method can be found (R-1), but since this is necessary to make the call, this check does not add any extra overhead.

Again, once the call returns, check T-2 ensures control will flow into a valid next instruction.

6.1.4 Return instructions

Finally, return instructions pop the return value from the stack, and then emit a native `RET` instruction to exit the method and return control, either directly to the AOT compiled code of the caller for lightweight method calls, or to the VM's `callMethod` function for normal methods.

The `RET` instruction takes the return address from the native stack, which is also used to store the VM's integer operand stack. This means the integer operand stack needs to be empty at return instructions (T-7) to ensure the correct address will be at the top of the native stack. Memory safety then guarantees the application could not have corrupted it. Without this check, malicious code could leave an integer on the stack and use the return instruction to jump to an arbitrary location.

A second way the return address could be corrupted is if the native stack overflows into the VM's heap. In CapeVM the heap is a fixed sized block that sits above other global variables, and below the native stack that grows down towards it. If the native stack grows into the area reserved for the heap, a return address may be corrupted by an otherwise valid heap write.

To prevent this, a run-time check is added to non-lightweight invokes, that checks the stack frame for the called method, plus its maximum integer stack size, does not grow into the heap (R-2). Lightweight calls do not add to the stack, since space for their local variables, stack, and return address was already allocated in the caller's frame.

While running, a method may make calls to the VM or `libc` functions, causing the native stack to grow further. Since the maximum stack growth for such calls is fixed, a certain safety margin is added between the stack and heap. A similar gap of 128 bytes between the stack and kernel heap is reserved by *t-kernel* [34], although it is not clear from the paper whether this is for the same reason.

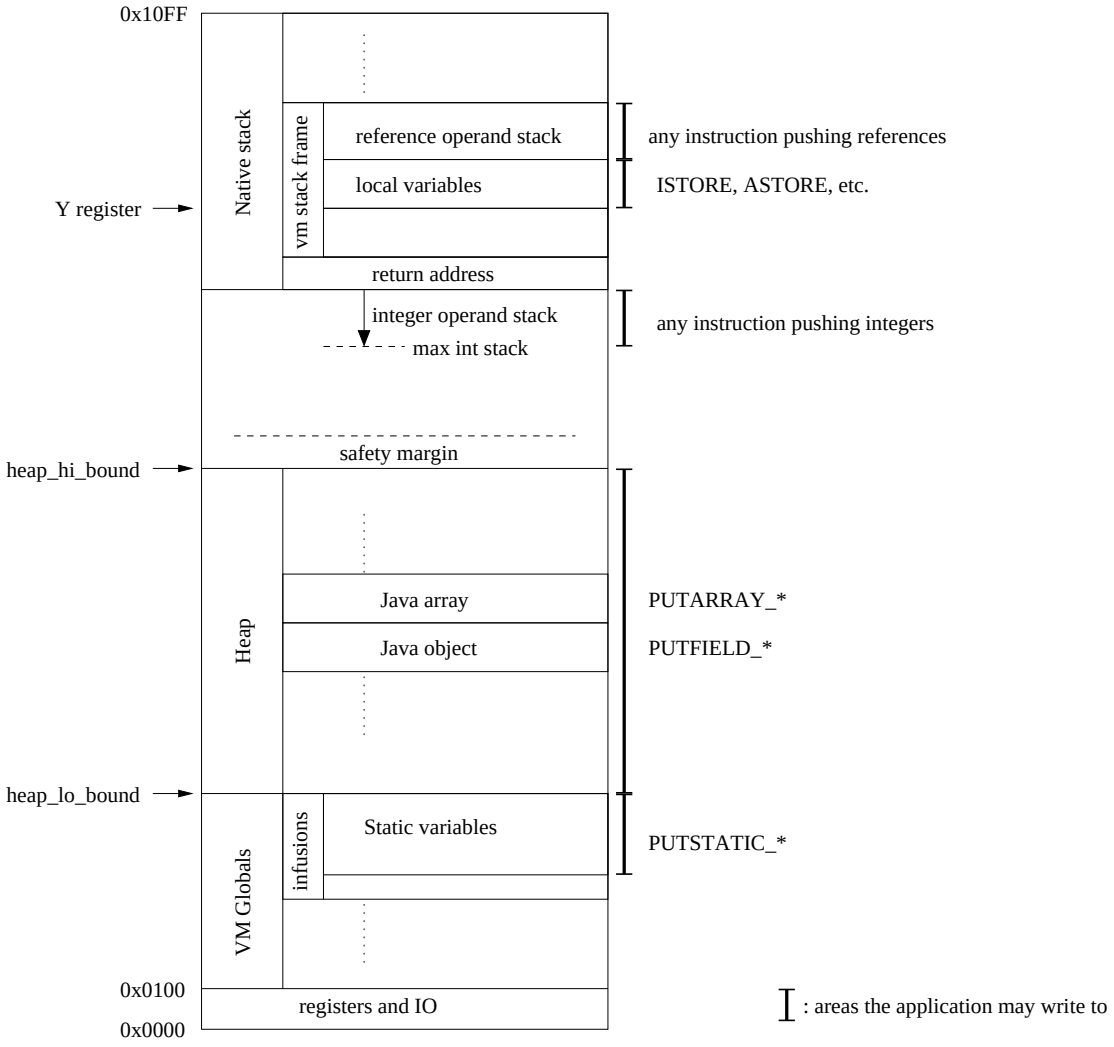

Figure 6.1: Global memory layout and the areas accessible to the application

Table 6.3: Instructions writing to memory

Type	Writes to
Any instruction pushing to the operand stack	The reference or integer stack
`STORE`	A local variable in the current method's stack frame
`PUTSTATIC`	A static variable in an infusion
`NEW`	The heap
`PUTARRAY`	An array on the heap
`PUTFIELD`	An object on the heap

6.2 Memory safety

Figure 6.1 shows the global layout of the VM's memory. At the bottom are the internal state of the VM, stored in a number of global variables, and a block with infusion descriptors which includes space for the Java static variables of each class in the infusion. This is followed by the application heap, which contains Java objects and arrays.

The native stack grows down in memory towards the heap. This contains a mix of native stack frames for internal VM functions, the application's VM stack frames containing space for local variables and the reference operand stacks, and the integer operand stacks which grow down directly on top of the native stack.

The VM's private data and the application data are mixed in the node's memory. The application is only allowed to write to the areas indicated with bars to the right. Any write outside of these designated areas may corrupt the VM's internal state, and needs to be prevented by safety checks.

Having ensured control flow safety, we can rely on the fact that the VM will always execute complete bytecode instructions, and the application cannot skip past inserted runtime checks. Similar to control flow safety, we demonstrate memory safety by grouping instructions with respect to their memory writes, as shown in Table 6.3, and defining the checks necessary for each category.

6.2.1 The operand stack

The VM will reserve space for the operand stacks based on the maximum stack depth in the method headers, so it needs to make sure the actual stack depth neither underflows, or

exceeds the maximum announced in the header. Everything in this section applies equally to the integer and reference stack.

The VM creates a stack frame based on the method header, so at translation time it is known exactly how much space will be available at run time. Lightweight methods do not create their own stack frame, but depend on the caller's stack frame. Whenever an `INVOKELIGHT` instruction is translated, the VM needs to verify the stack frame of the current method has reserved enough free space for the lightweight method's stack (T-8).

The stack effect of each instruction is known at translation time, so the stack depth can be verified in a single top to bottom pass. While translating a method, the VM maintains two counters indicating how many values are on the integer and reference stacks, and updates these counters for each instruction's stack effects. For normal methods both counters are initialised to 0, since they start with empty stacks. Lightweight methods start with their parameters on the stack, so for these the counters are initialised according to the number of arguments announced in the method header.

For each translated instruction, the VM checks there are enough values on the stack to consume its operands (T-9), and that maximum stack depth announced in the header is not exceeded after pushing its results (T-10).

Branches The `BRTARGET` instruction poses a problem for this single pass approach since it can be reached from multiple locations, so the stack depth when entering this instruction is unknown.

The simplest way to solve this is to require the stack to be empty at all branches. As mentioned in 5.3.2, this is already the case for most code generated by `javac`.

Alternatively, the expected stack state at each branch target could be included in the method header, which would allow the VM to check the state matches this expected state at each branch and branch target. However, this is more complex and in practice the overhead for requiring an empty operand stack at branches is minimal: in our entire Java codebase only a few small modifications were necessary, which a future combined optimising infuser could do automatically.

Therefore, the VM only verifies the stack is empty at branches and branch targets

(T-11).

Invoke instructions Most bytecode instructions have a fixed effect on the stack, for example `IADD` will always consume two 32-bit ints and push another. We encode this in a simple table. Method calls require some more attention.

The `INVOKESTATIC`, `INVOKESPECIAL`, and `INVOKELIGHT` instructions all contain the id of the method that will be invoked. For these, the number of arguments and return type in the target method's header are used to determine the instruction's stack effects.

For `INVOKEVIRTUAL` and `INVOKEINTERFACE` the actual method that will be called depends on the object on the stack at run time. For these, the expected stack effect is determined based on the first implementation that matches the call. For valid code all implementations should have the same signature, and thus the same effect on the stack, but malicious code could send an implementation in a subclass that has different stack effects. Therefore, a run-time check (R-3) is added that verifies the method called at run time has the same stack effects as the one used to verify the stack at translation time.

Return instructions Note that `RETURN` instructions do not need any special care. The stack depth in the method is verified using the instruction found in the bytecode. It is possible for a method to break the contract established in the method header, for example by using `RETURN` instead of `IRETURN` in a method that should return an int. However, this is still safe as long as the stack is empty after the return instruction, as checked by T-7.

Because the return value is passed back to the calling method in registers, the result of using an incorrect return instruction is that either the return value is discarded, or whatever happens to be in the registers is used as a return value, which may corrupt the application's own state, but not the VM's.

6.2.2 `STORE`

Local variables are accessed as an offset from the Y register, which is under control of the VM and points to the start of the local variables, as shown in Figure 6.1. Similar to

the operand stack, the method header contains the number of variable slots that will be allocated in a normal method's stack frame. For lightweight methods, the VM checks `INVOKELIGHT` instructions to verify the caller has reserved enough local variable slots for the target method (T-12), so the number of slots specified in a lightweight method's header is also guaranteed to be available at run time.

Local variables are written to using the `STORE` instructions. Each `STORE` instruction contains the index of the local variable slot to write to, so the VM only needs to check at translation time that the index of the local is within the range announced in the method header to guarantee it writes to a valid location (T-13).

6.2.3 `PUTSTATIC`

Static variables are allocated globally at the start of the application based on number of static variables in the *infusion* header. The `PUTSTATIC` instruction contains the id of an infusion, and the index of the target static variable slot. At translation time, the VM checks the referenced infusion exists (T-14), and the index is within the legal range (T-15) for that infusion.

6.2.4 `NEW`, `PUTFIELD` and `PUTARRAY`

The final type of memory access is to the heap. The various `NEW` instructions used to create arrays and objects are fully implemented by a call to the VM, which only allocates new objects at a valid heap location and will terminate the application if it runs out of memory. Writes to object fields and array elements happen using the `PUTFIELD` and `PUTARRAY` instructions.

These instructions both work on an object reference, so a null reference bug could easily cause the VM to write to the lowest addresses. In the ATmega, the lowest 32 bytes of the address space are mapped to the CPU's general purpose registers, so this can cause very hard to diagnose bugs. Similarly, using a high out-of-bounds index into an array, malicious code could gain access to the native stack and, for instance, corrupt return addresses.

In some cases it may be possible to verify these operations at translation time, but

this is hard without extensive analysis that would be too expensive for a sensor node. Therefore a run-time check is added when translating these instructions, which checks the target address is within the heap just before the actual write to memory (R-4).

```
1     heapcheck:
2         lds  r0, heap_lo_bound
3         cp   ZL, r0
4         lds  r0, heap_lo_bound + 1
5         cpc  ZH, r0
6         brlo illegal_access_handler:
7         lds  r0, heap_hi_bound
8         cp   r0, ZL
9         lds  r0, heap_hi_bound + 1
10        cpc  r0, ZH
11        brlo illegal_access_handler:
12        ret
```

Listing 9: Heap bounds check

The VM stores the bounds of the heap in two variables: `heap_lo_bound` and `heap_hi_bound` as shown in Figure 6.1. Each heap access instruction calculates the address to write in the ATmega's Z register. Just before the write to the heap, the VM inserts a `CALL` to the `heapcheck` function shown in Listing 9. This function checks the address in Z is within these bounds. If not, it jumps to the `illegal_access_handler`, allowing the VM to terminate the application. This adds 22 cycles overhead for each array or object write, and 4 bytes code size overhead for the `CALL` instruction.

The actual write to the heap is often done by an offset from Z using the AVR's `STD`, or 'store indirect with displacement' instruction, that allows writes to a fixed offset of at most 63 bytes from Z. For example, to write to object fields who's offset within the object is known at translation time, the VM simply loads the object's address into Z and uses `STD` to write to the correct offset.

This means the write target could an address at most 63 bytes above the end of the heap. One way to avoid this is to avoid the `STD` instruction, and instead use the `ADIW` instruction to first add the offset to Z and then use the normal `ST` instruction to store without displacement. However, this would add an overhead of 2 bytes and 2 cycles. Instead we reuse the same small safety margin mentioned in Section 6.1.4 for check R-2. Reusing the same margin for both cases is safe because the VM can never write to a heap object and execute a function in the VM or *libc* at the same time.

Alternatives We considered several alternative implementations for `heapcheck`. Since the `CALL` and `RET` instructions are expensive, 7 cycles can be saved by inlining the check instead of calling it. However, this increases the code size overhead from 4 bytes to over 30 bytes, which we consider too high.

If the top and bottom boundaries of the heap are aligned at 256 bytes, this eliminates the need to check the lower byte of the Z register. This saves 6 of the 22 cycles, but wastes RAM since some bytes below and above the heap would have to remain unused. Since RAM is such a scarce resource and the performance gain is limited, we decided against this.

Finally, 8 cycles can be saved by keeping the bounds in registers instead of memory, which removes the need for the `LDS` instructions. However this reduces the performance of the stack cache since these registers would not be available for stack caching. Which of these affects performance more depends on the code being executed. We evaluate the difference in Section 7.8.1. Since having the bounds in registers is more complex to implement, thus increasing VM size, we choose to keep the bounds in registers.

Heap corruption Since the garbage collector compacts the heap when it frees any memory, the heap is always split into an in-use part and a free part. A pointer in the VM points to the first free byte. This pointer is moved forward when memory is allocated, and moved backward when the garbage collector compacts the heap.

The in-use part of the heap is made up of chunks, which have a small header indicating their size. The heap access check only verifies a write is to an address within the heap, but not that the address is a correct address within the target object. Code may still corrupt parts of the heap, including these heap headers.

This does not break the safety guarantees since these heap headers are not used until control is returned to the VM. The headers are only used by the garbage collector, so before the garbage collector starts, the VM must check the intergrity of the heap headers (R-5), which must form a consistent chain of chunks ending at the first free byte.

Table 6.4: Comparison of CapeVM's safety guarantees to source code approaches

	CapeVM and native code approaches	Source code approaches
Safety checks added/verified at	The node	The host
Protects	The VM	The VM and application
Protects against malicious code	Yes	No
Detects certain programming errors	No, but could be added at a cost	Yes

6.3 Comparison to other systems

This section compares CapeVM's approach to other systems providing safety. Section 4.10 of the Java Virtual Machine Specification [56] specifies a number of checks an implementation must do to comply with the standard. CapeVM's checks are different in two ways: First, they are defined at a lower level, specific to our VM's implementation. Second, since CapeVM's goal is only to ensure the application cannot corrupt the VM, its checks are less restrictive than the JVM specification.

For example, CapeVM allows out of bounds array indexes. While incorrect, these do not violate the safety guarantees as long as the write stays within the heap. Out of bounds array access indicates a bug in the programme, and failing early instead of corrupting the application's state usually makes it much easier to find the bug. CapeVM's checks ensure malicious code cannot corrupt the VM, but they do not prevent such programming errors from corrupting the application's own state, unless they also violate the safety constraints.

Section 3.7.1 introduced a number of systems such as Safe TinyOS [19] that work on the source code level. While seemingly similar, they are actually the reverse in the sense that their more fine-grained checks do prevent certain programming errors from corrupting the application's state, but since the node assumes the necessary checks to be in place, they cannot guard against malicious code sent to the device. Because in these systems the safety checks are added by the host, more complex analysis can be done on the source code to prove certain operations to be safe at compile time, which reduces the number of necessary safety checks and thus the run-time overhead.

The checks in the JVM's specification both protect against malicious code and detect certain programming errors. Desktop JVMs, like the host in Safe TinyOS, have ample

resources to do the analysis necessary to reduce the overhead of fine-grained checks. On a sensor node we argue the two goals require two separate solutions.

While CapeVM's checks could be extended to provide the same level of safety as Safe TinyOS, to do so would require extending the existing checks to include the size of the object or array that is being accessed, which would make them considerably more expensive. Since CapeVM adds the safety checks on the node, it lacks the resources necessary to do the analysis that allows Safe TinyOS to eliminate many checks. Instead, CapeVM must conservatively check each access.

If the goal is to protect the application from programming errors that could corrupting its own state, this implies control over the code, and an approach similar to that of Safe TinyOS will be more efficient. Safe TinyOS works on native nesC code, but a similar system could be developed for Java where the checks inserted by the host are implemented as new bytecode instructions to mark array or object accesses than need run-time checking.

The difference between both approaches is summarised in Table 6.4. For some applications both may be useful. For example, in a system like the Amulet smart watch, CapeVM's safety checks are useful to isolate an untrusted application from the VM. At the same time, more fine-grained checks may be added to detect buggy applications. In this case, adding these checks on the host, similar to Safe TinyOS, will lead to less overhead compared to extending those added by the node.

6.3.1 SensorScheme

Next, we will compare the approach taken by CapeVM to three existing systems that provide safety on the node. First, SensorScheme is the only sensor node VM to explicitly mention safety, although it does not describe many the details.

SensorScheme is a LISP dialect, so both code and data are stored as lists. In SensorScheme memory is organised as a collection of fixed-sized *cells* that make up these lists. Since the cells are managed by the VM this inherently gives it a level of safety, and run-time checks are added to check the datatypes and length for each operation. Since it is an interpreter, it has a large run-time overhead, which makes the added overhead of these

checks insignificant.

6.3.2 *t-kernel*

In addition to safety, *t-kernel* [34, 35] also provides a form of virtual memory which makes it hard to compare to CapeVM directly. The authors name OS control and memory integrity as the two primitives the system needs to provide in order to protect the kernel from the application.

OS control and control flow safety OS control is defined as the ability of the OS to take control of the CPU. The authors note that "Traditionally, the CPU control is guaranteed by privilege support and clock interrupts. However, many microcontrollers used by sensor nodes do not have privilege support. The application can disable interrupts and occupy the CPU for an arbitrarily long time."

To implement its virtual memory features and safety guarantees, *t-kernel* extensively rewrites, or *naturalizes*, the application's binary code at run time, on demand, one 256-byte page at a time. As a result, addresses in the original binary, do not match the physical addresses in the naturalized pages. This is solved by replacing all branching instructions, including calls, returns, etc, with calls back to the kernel, which then looks up the corresponding physical address.

This causes a slowdown of up to 30x. To reduce this overhead, for forward branches these calls back to the kernel are replaced with a direct jump after the first transition occurs. Backward branches are replaced with code that first increment an 8-bit counter, and calls back to the kernel when this reaches zero. This guarantees the kernel gets back control at least once for every 256 backward branches, without depending on a timer, but at the cost of a slight overhead. Control flow safety is not explicitly discussed, but is guaranteed by the same mechanism. Since each branch targets a virtual address, which is translated to a physical address by the kernel, the kernel can ensure the physical address is correct.

Compared to *t-kernel*, it is easier for CapeVM to guarantee it can regain control of the CPU. Since the VM offers no functions for the application to turn of interrupts or modify

timers, it *can* simply use an interrupt to ensure it periodically regains control of the CPU.

Memory safety Memory in *t-kernel* is split into three regions: (i) physical addresses that map to IO ports, special registers, etc, (ii) the stack, and (iii) the heap. Most accesses, to local variables, parameters, register saves and restores, etc are to the stack. The naturalization process ensures stack access is safe, although the paper does not go into details. Stack access is optimised, but there is some overhead for certain types of access. In the worst case, indirect addressing with offset, this takes 5 instead of 2 cycles. In CapeVM, local variable access incurs no extra overhead from safety checks, since it can determine at translation time whether or not it targets a valid local variable slot.

t-kernel's virtual memory provides the application with a 60 KB heap. This is divided in 16 byte pages, some of which are kept in buffers in RAM. Each buffered page has a header indicating its virtual memory starting address. When the application accesses the heap, *t-kernel* searches the pages in RAM, starting with the one used for the most recent access. If the required page is not in RAM, it is swapped in from flash. Besides providing virtual memory, this also guarantees safety since a heap write always writes to a buffered page, and the entire virtual heap is assigned to the application.

Access to the heap is both considerably slower than stack access, and highly variable. The paper lists a benchmark which does repeated heap accesses without swapping. This takes 16 cycles, compared to 2 for a normal memory access. The paper does not mention whether the benchmarks targets different pages, but the low overhead would suggest it may target the same address for every write, which means the first page the kernel examines is the correct one. If the required page is not found in RAM, it is swapped in from flash, which takes over 180,000 cycles.

The best case access of 16 cycles is comparable to CapeVM's 22 cycle overhead for heap access, but will rise if multiple pages need to be searched by the kernel, or if pages need to be swapped in from flash. Unfortunately the paper does not mention how many pages are buffered in RAM, but it is clear applications with more data - the ones that would benefit from having a larger virtual memory - will incur a higher overhead, either from having to scan more buffered pages, or from having to do more swapping.

In conclusion, *t-kernel*'s performance overhead should be slightly higher than CapeVM's in most cases. In addition, the extensive rewriting of the binary code adds a large code size overhead, reported at a 6-8.5x increase.

6.3.3 Harbor

Harbor guarantees safety by adding checks on the host. A verifier on the node then verifies all checks are in place before executing the programme. While checks are added by the host with its ample resources, Harbor is still bound by the limited resources on the node since the node needs to be able to verify correctness.

Control flow safety In a Harbor application, run-time checks are added to protect writes. To guarantee applications cannot jump past these run-time checks, Harbor disallows computed branches so the verifier can check the target address of each branch.

Function returns could also be used to jump to an arbitrary location if the return address can be corrupted. To prevent this, Harbor uses function entry and exit stubs that store the return address in a reserved 'safe stack' in a reserved section of memory not accessible to the application. This adds an overhead of 76 cycles per function call.

Memory safety Memory safety in Harbor differs significantly from CapeVM. The entire address space is split into fixed sized blocks, each of which can be assigned to a module. A memory map keeps track of the ownership of each block.

Harbor supports multiple modules, and both the block size and the maximum number of modules are parameters that can be changed. The overhead for storing the memory map is 256 bytes for an 8-byte block size and maximum of 8 modules, which are the defaults used in the paper. Using only 2 modules, which is sufficient to isolate the OS and the application, this is reduced to 128 bytes. Increasing the block size will also reduce the size of the memory map, but at the cost of greater fragmentation.

All memory writes are preceded by a call to the `write_access_check` function, which checks the current module has permission to write to the target address. This imposes a run-time overhead of 65 cycles per write.

CapeVM only needs a run-time check for writes to the heap, while Harbor checks *all* writes, including local variables. This, combined with Harbor's more expensive `write_access_check` function, suggest its overhead will be significantly larger than CapeVM's.

Verifier Although Harbor's safety checks are added on the host, the correctness of the system only depends on the verifier running on the node. This verifier is a relatively small and simple component.

This is a significant advantage of Harbor's approach. Malicious attacks often exploit bugs in the system they are trying to corrupt. Compared to more complex systems like *t-kernel* and CapeVM, the simplicity and small size of Harbor's verifier reduces the chance of exploitable bugs, but this comes at the cost of an increased run-time overhead.

Chapter 7

Evaluation

This dissertation presents a number of techniques to improve the performance of sensor node virtual machines and make them safe, while staying within the constraints set out in Section 4.1. This chapter evaluates to what extent CapeVM meets these goals by measuring its performance and code size overhead for a number of different benchmarks.

First, Section 7.1 describes our experimental setup, the benchmarks used, and how the source code for these benchmarks was obtained.

Next, Section 7.2 uses the largest benchmark to examine the effect of the lack of optimisations done by the standard `javac` compiler, and the manual optimisations performed on the Java source that in a future version should be done by an optimising infuser.

Sections 7.3, 7.4 and 7.5 evaluate the result of the optimisations to the AOT translation process on performance and code size.

Sections 7.6 and 7.7 focus on two specific optimisations: adding support for constant arrays and lightweight method calls.

The cost of adding safety checks is examined in Section 7.8, which also compares CapeVM's overhead to existing native code systems that provide safety.

Platform independence is one of the main reasons to use a VM. While CapeVM was only implemented for the ATmega128, Section 7.9 presents measurements that give an indication of the expected performance on other common sensor node platforms.

Finally, in Section 7.10 we discuss the limitations and cost of using a VM, and describe some known hard cases our VM currently does not handle as well.

Table 7.1: Benchmarks used in the evaluation

Benchmark	Source and input data	Size	Typical sensor node code	Used as a benchmark in
Bubble sort	Darjeeling sources [12] *Input:* 256 16-bit numbers sorted in reverse order, as in the original source.	single function	no	[13, 24]
Heap sort	Standard heap sort taken from [3] *Input:* 256 16-bit numbers sorted in reverse order.	two functions	no	
Binary search	TakaTuka sources [6] *Input:* worst case (not found) search in 100 16-bit values, as in [24].	single function	no	[24]
XXTEA	Wheeler and Needham [101] *Input:* 32 32-bit numbers. Contents do not affect performance.	single function	no	
MD5	Darjeeling sources [12] *Input:* the string 'message digest' as in the original source.	single function	no	[13, 24]
RC5	LibTomCrypt [85] *Input:* first test case in libtomcrypt sources (64 bit)	single function	no	
FFT	Fixed point FFT using the widespread `fix_fft.c` [93] *Input:* 64 8-bit or 16-bit numbers. Contents do not affect performance.	two functions	yes	[49]
Outlier detection	Our implementation of the algorithm described in [49] *Input:* 20 16-bit values increasing from 0 to 19, with outliers of 1000 and -1000 at index 2 and 11.	single function	yes	[49]
LEC	Our implementation of the compression algorithm described in [63] *Input:* 256 16-bit ECG measurements downloaded from PhysioNet [71].	three functions	yes	
CoreMark 1.0	EEMBC [88] *Input:* defined in CoreMark source.	full application	no	
MoteTrack	Lorincz [60, 59] *Input:* defined in MoteTrack source.	full application	yes	
Heat detection	Adapted from code used in our group to track objects using an 8x8 pixel heat sensor *Input:* 101 frames of 8x8 16-bit values for calibration, 25 frames for detection.	full application	yes	

7.1 Benchmarks and experimental setup

This section describes our experimental setup, the benchmarks used and how the source code for each benchmark was obtained, and any relevant details in their implementation.

A set of twelve different benchmarks, shown in Table 7.1, is used to measure the effect of the optimisations and the overhead of safety checks. This mix of benchmarks was chosen for several reasons. A number of benchmarks, *bubble sort*, *binary search*, *MD5*, *FFT*, and *Outlier detection* were chosen because they are used in various related work, allowing a comparison of CapeVM to these results.

A number of benchmarks are small benchmarks that process arrays of data. While the actual processing done may not be typical for sensor networks (although the *MoteTrack* application does do a bubble sort), the small size of these benchmarks make them useful to highlights specific behaviours that would be lost in the averages of a larger benchmark.

The *CoreMark* benchmark is an industry standard benchmark to measure CPU performance. It is a larger benchmark, mixing several kinds of processing: besides array processing in the form of matrix operations, it also contains linked list processing and a state machine. Since *CoreMark* mixes different kinds of processing, it is a good example of the expected average behaviour. The many different methods enables us to evaluate the effect of method calls and show CapeVM can effeciently handle larger, more complex applications.

Finally, *Outlier detection*, *LEC*, *MoteTrack* and *heat detection* are all code that was specifically developed for sensor nodes, and *FFT* is a typical signal processing operation, which is a common and potentially expensive task for sensor nodes.

Sensor nodes spend their time and energy on three main tasks: accessing sensor and actuators, communication, and data processing. The first two require interaction with the hardware, so they must be implemented in native code in the VM's standard library. Since native code is not influenced by the performance of the VM, the benchmarks used in this evaluation focus on data processing.

As noted before, to what extent an application is affected by the VM's slowdown is highly application dependent. The Amulet smart watch system discussed in Chapter 1

notes that energy consumed when the CPU is in active mode is significant, and their breakdown of the CPU active time shows most is spent in application code rather than the OS. Similarly, the Mercury motion analysis platform show the energy spent on feature extraction or FFT becomes significant or dominant in the total energy consumption when multiplied by the typical slowdown seen in interpreting VMs.

We argue that some form of array processing will be common in many sensor node applications, and especially so in applications that are significantly affected by the VM's slowdown. First, arrays of data appear in many sensor node application, both in the form of sensor data, and as sent or received radio messages that need to be constructed or parsed. Second, processing such arrays is likely to be a significant part of the total processing time, for the simple reason that looping over an array of elements quickly takes more time than processing a single value.

Finally, we note that compared to benchmarks on more complex high performance CPUs, the benchmarks here are less affected by the exact workload. The simple CPUs found on sensor nodes typically have no caches, and no (ATmega and MSP430) or very short (Cortex M0) pipelines. Thus, factors like branch prediction and cache line alignment that can have a very large impact on larger systems, have no impact on the results presented here. The largest performance difference found in all benchmarks is between a 1.18x slowdown for FFT, and 2.56x slowdown for MoteTrack.

7.1.1 Implementation details

To ensure the results can be reproduced, we describe the implementation of our benchmarks in this section. For most benchmarks a C version is available in the sources mentioned in Table 7.1. The sources for *heat detection*, *LEC* and *outlier detection* are not available online, but are listed in the appendices.

The *bubble sort*, *heap sort*, *FFT*, *binary search*, and *outlier detection* benchmarks could all be implemented for different data sizes. In this evaluations 16-bit data is used. 8-bit data is too small for many tasks, for example the ATmega's has 10-bit ADCs, and the memory constraints of sensor nodes mean developers will often be reluctant to use 32-bit

variables where 16 bits are sufficient. Therefore, the middle option is used for the main evaluation, and the effect on the performance of 8-bit or 32-bit data is discussed in Section 7.9.

The C versions of these benchmarks were first translated directly to Java, keeping both implementations as close as possible. The result was then manually optimised as described in Section 5.2. These optimisations did not affect the performance of the C version significantly, indicating `avr-gcc` already does similar transformations on the original code.

There are cases where a developer who is aware of the performance characteristics of the VM may choose a different approach than the one used in the C version when developing a Java implementation directly. We discuss some of the issues when translating C to Java for the *CoreMark* benchmark in Section 7.2.2, including two changes that lead to better performance. The C version was followed as closely as possible to avoid bias by optimising specifically for our VM. We take a bit more liberty for the *MoteTrack* and *heat detection* benchmarks, since these could not be directly translated.

The benchmarks exposed some limitations of using a VM instead of native code, which are common to most sensor nodes VMs. Specifically, the lack of support for constant data, high memory overhead for code containing many small objects, and high performance overhead for allocating temporary objects. These are discussed in more detail in Chapter 8, where we also suggest options to solve some of these limitations.

FFT

Both 8-bit and 16-bit versions of the `fix_fft.c` implementation exist. In the main evaluation we use the 16-bit version taken from the Harbor source code [93].

Both versions contain a table of precalculated sine wave values, which are stored in flash using the constant array optimisation introduced in Section 5.3.5.

Outlier detection

We implemented the outlier detection algorithm as described in [49]:

> ”The outlier detector samples a set of sensor values and stores them in a buffer. Once the buffer is filled, it computes the distance between all pairs of samples in the buffer and stores the result in a matrix. Using a distance threshold, the algorithm marks the distance measurements in the matrix that are greater than the threshold. If the majority of the distance measurements for a sensor readings are marked, then the sensor reading is classified as an outlier.”

Note that there is no reason to store the distances in a matrix, and having to allocate this matrix limits us to only small arrays of input data. The same result can be calculated directly, without the distance matrix, by examining the samples one at a time, and counting the number of other samples with a distance higher than the threshold.

Because this benchmark will be used to compare CapeVM’s safety cost to Harbor’s, we implemented it as described in the paper.

LEC compression

The LEC algorithm is described in detailed pseudo code in [63]. Our implementation follows this pseudo code as closely as possible, and is listed in Appendix A. The input is a set of 256 16-bit ECG measurements downloaded from PhysioNet [71].

MoteTrack

The *MoteTrack* application uses received signal strength (RSSI) measurements from a number of beacon nodes to do indoor localisation. It contains a database in flash memory of reference RSSI signatures, stored in a complex structure of many small structs and arrays in C.

The memory overhead when translating this directly to Java was too high to run the application, forcing us to make two modifications. First, the original source has the option to list RSSI signatures at different transmission powers, but the authors note this may not always improve results. The original C code only uses a single transmission power setting, which results in arrays of a single element that get optimised away at compile time. In the Java version these were replaced with simple variables. Second, a two element array with

RSSI values for different channels was flattened into two separated variables to eliminate the overhead from allocating too many small arrays.

Again, the constant array optimisation was used to place the data in flash memory. Since this only allows arrays of integer types, the single array of C nested structs was split into 7 arrays for the individual fields. Without this optimisation, it would be impossible to implement this application in Java because the 20 KB database is too large to fit in RAM.

Thus, while our Java implementation of *MoteTrack* does execute the same algorithm as the C version, we were force to modify its implementation significantly, which clearly highlights some of the weaknesses of current sensor node VMs. These changes do not affect the results of the current version of the code, but we note that while it would be possible to use multiple transmission powers or more channels in the C version, this would require too much memory for the Java version.

Heat detection

The *heat detection* application is adapted from code used by a different project in our group to track persons and fire hazards using Raspberry Pi devices equipped with an 8x8 heat sensor.

It contains two phases: first the heat sensor is calibrated with no heat sources in view to determine the average and standard deviation of the sensor readings. In the next phase the algorithm tracks the position of a person moving within the field of view of the sensor, and detects extreme temperatures that may indicate a fire. In the evaluation both phases are listed separately.

The calibration phase was modified to allow it to run on the more resource-constrained sensor node, but the results of the calibration are identical. The code for the detection phase was copied directly from the source used on the Raspberry Pi, but modified slightly to avoid repeatedly allocating temporary objects as described in Section 8.8.

Our implementation reads sensor measurements using a native call to read from a table in flash memory, simulating to how a real version would use a library call to read from a sensor.

7.1.2 Experimental setup

Each benchmark is implemented as a C and a Java version. We compile these using `javac` version 1.8.0, ProGuard 5.2.1, and `avr-gcc` version 4.9.1. The C benchmarks are compiled at optimisation level -O3, the rest of the VM at -Os.

A manual examination of the compiled code produced by `avr-gcc` revealed some points where more efficient code could have been generated. But with the exception of the lack of some constant shift optimisations discussed in Section 5.3.5, these did not affect performance by more than a few percent. This leads us to believe `avr-gcc` is a fair benchmark to compare to.

Each benchmark was run in the cycle-accurate Avrora simulator [90], emulating the ATmega128 processor. Avrora was modified to emit traces of the AOT translation process and of the run-time performance. During AOT translation, the VM writes to a specific memory address monitored by Avrora to inform it of each step in the process. When running both the C and AOT compiled benchmarks, Avrora tracks the number of cycles spent in each instruction. These traces, combined with debug output from the infuser and disassembled native code provide a detailed view of the performance on a per-instruction basis.

The main measure for both code size and performance is the overhead compared to optimised native C. To compare different benchmarks, this overhead is normalised to a percentage of the number of bytes or cpu cycles used by the native implementation: a 100% overhead means the AOT compiled version takes twice as long to run, or twice as many bytes to store. The exact results can vary depending on factors such as which benchmarks are chosen, the input data, etc., but the general trends are all quite stable.

7.2 CoreMark

The *CoreMark* benchmark was developed by the Embedded Microprocessor Benchmark Consortium as a general benchmark for embedded CPUs. It consists of three main parts:

- Matrix multiplication

- A state machine

- Linked list processing

Since *CoreMark* is the largest benchmark, we will use it to discuss some of the challenges when translating its C code to Java.

The biggest complication is that *CoreMark* makes extensive use of pointers, which do not exist in Java. In cases where a pointer to a simple variable is passed to a function, we simply wrap it in a wrapper object. A more complicated case is the `core_list_mergesort` function, which takes a function pointer parameter `cmp` used to compare list elements. Two different implementations exists, `cmp_idx` and `cmp_complex`. Here we initially choose the most canonical way to do this in Java, which is to define an interface and pass an object with the right to implementation to `core_list_mergesort`.

The C version of the linked list benchmark takes a block of memory and constructs a linked list inside it by and treating it as a collection of `list_head` and `list_data` structs, shown in Listing 10. One way to mimic this as closely as possible is to use an array of shorts of equal size to the memory block used in the C version, and use indexes into this array instead of C pointers. However this leads to quite messy code.

Instead we choose the more natural Java approach and define two classes to match the structs in C and create instances of these to build the list. This is also the faster option because accessing object fields is faster than array access. The trade-off is memory consumption, since each object has its own heap header.

```c
typedef struct list_data_s {
    ee_s16 data16;
    ee_s16 idx;
} list_data;

typedef struct list_head_s {
    struct list_head_s *next;
    struct list_data_s *info;
} list_head;
```

```java
public static final class ListData {
    public short data16;
    public short idx;
    }

public static final class ListHead {
    ListHead next;
    ListData info;
}
```

Listing 10: C and Java version of the CoreMark list data structures

Table 7.2: Effect of manual source optimisation on the CoreMark benchmark

	list		matrix		state		total	
	time [a]	vs nat. C	time [a]	vs nat. C	time [a]	vs nat. C	time [a]	vs nat. C
native C	17.9		49.2		18.0		85.1	
baseline	124.3	(+594%)	360.5	(+633%)	289.0	(+1506%)	774.4	(+810%)
optimised, using original source	55.9	(+212%)	231.4	(+370%)	75.9	(+322%)	363.4	(+327%)
manually inline small methods	-0.8	(-4%)	-37.3	(-76%)	-17.4	(-97%)	-55.5	(-65%)
use short array index variables	-0.1	(-1%)	-104.6	(-213%)	0.0	(0%)	-104.7	(-123%)
avoid recalculating expressions in a loop	+0.1	(+1%)	-7.6	(-15%)	0.0	(0%)	-7.5	(-9%)
reduce array and object access	-0.1	(-1%)	-18.0	(-37%)	-2.4	(-13%)	-20.5	(-24%)
reduce branch cost in crcu8	-3.6	(-20%)	-0.5	(-1%)	-3.2	(-18%)	-7.3	(-9%)
using optimised source	51.4	(+187%)	63.4	(+29%)	52.9	(+194%)	167.9	(+97%)
(non-autom.) avoid creating objects	0.8	(+4%)	0.0	(0%)	-10.6	(-59%)	-9.9	(-12%)
(non-autom.) avoid virtual calls	-22.8	(-127%)	0.0	(0%)	0.0	(0%)	-22.7	(-27%)
after non-automatic optimisations	29.4	(+64%)	63.4	(+29%)	42.3	(+135%)	135.3	(+59%)

[a] in millions of cycles

7.2.1 Manual optimisations

After translating the C code to Java, we do some manual optimisations to produce better bytecode. Since *CoreMark* is the most comprehensive benchmark, we use it to evaluate the effect of these manual optimisations.

Table 7.2 shows the slowdown over the native C version, broken down into *CoreMark*'s three main components. The baseline version, using the original Java code and without any optimisations, is 810% slower than native C. Even after applying all our other optimisations, the best we can achieve with the original code is a 327% slowdown.

Next we manually optimise to the Java source code, starting with the optimisations as described in Section 5.2 and add a small extra optimisation to `crcu8` which can be easily reorganised to reduce branch overhead. These are all optimisations that a future optimising infuser could do automatically.

The effect depends greatly on the characteristics of the code. The matrix part of the benchmark benefits most from using short array indexes, the state machine frequently calls a small method and benefits greatly from inlining it, etc. Combined these optimisations reduce the overhead for the whole benchmark from 327% to 97%, proving the importance of a better optimising infuser.

We also applied all these optimisations to the native C version to ensure a fair comparison, but the difference in performance was negligible.

7.2.2 Non-automatic optimisations

After these optimisations, *CoreMark* is still one of the slower benchmarks. We can improve performance further using two more optimisations. While these cannot be done automatically, even by an optimising infuser, they do not change the meaning of the programme, and a developer writing this code in Java from the start may make similar choices to optimise performance.

Table 7.2 shows that in the native version, over half of the time is spent in the matrix part of the benchmark, but for the final Java version all three parts are much closer together. The state machine and linked list processing both suffer from a much larger slowdown than the matrix part, which by itself would be the third fastest of all our benchmarks.

One of the reasons for the slow performance of the state machine is that it creates two arrays of 8 ints, and an little wrapper object for a short to mimic a C pointer. Allocating memory on the Java heap is much more expensive than it is for a local C variable. For the linked list part the biggest source of overhead is in the virtual method call to the compare objects in `core_list_mergesort` that was used instead of a function pointer. Virtual methods cannot be made lightweight.

This is the best we can do when strictly translating the C to Java code, using only optimisations that could be done automatically. If this constraint is relaxed, these two sources of overhead can be removed as well: we can avoid having to repeatedly create the small arrays and objects in the state machine, by creating them once at the beginning of the benchmark and passing them down to the methods that need them. This significantly speeds up the state machine, although the list processing part incurs a small extra overhead because it needs to pass these temporary arrays and objects to the state machine.

The virtual call to the comparer objects in the list benchmark is the most natural implementation this in Java, but since there are only two implementations, we can make both compare methods `static` and pass a boolean to select which one to call instead of the comparer object. This saves the virtual method call, and allows ProGuard to inline the methods since they are only called from a single location.

Combined, this improves the performance of *CoreMark* to only 59% overhead over

native C, right in the middle of the spectrum of the other benchmarks.

Similar to *MoteTrack* in the previous section, these results point at some weaknesses of Java when used as an embedded VM. The lack of cheap function pointers, or a way of allocating small local objects or arrays in a method's stack frame means there will be a significant overhead in situations where the optimisations used here cannot be applied. We discuss a way to reduce the cost of such temporary objects in future VMs in Section 8.8.

In the rest of the evaluation, all results presented are for the manually optimised code. For *CoreMark* this includes the two non-automatic optimisations. The optimisation to avoid repeatly creating temporary objects was also applied to the *LEC* and *MoteTrack* benchmarks.

Table 7.3: Key benchmark characteristics, using optimised source code

	B.sort	H.sort	Bin.Search	XXTEA	MD5	RC5	FFT	Outlier	LEC	CoreMark	MoteTrack	HeatCalib	HeatDetect	average
CODE SIZE (BYTES)														
Bytecode	74	134	83	379	2983	453	441	287	334	2788	2552	310	2661	
Native C	118	298	146	1442	9458	910	1292	380	560	6128	3906	1944	5294	
AOT original	418	1012	412	3792	29502	4090	2576	1402	1628	13982	12784	2454	17248	
AOT optimised	258	596	310	2236	14654	2018	1324	800	1056	8990	8478	1610	10346	
EXECUTED BYTECODE INSTRUCTIONS (% of total executed bytecode instructions before optimisation)														
Load/Store	79.8	71.7	58.1	44.9	43.3	41.1	61.1	69.0	59.5	54.1	70.3	51.8	48.0	57.9
Constant load	0.2	8.1	11.0	12.5	19.1	17.6	6.4	0.6	7.9	10.0	5.4	10.1	16.6	9.7
Processing	8.0	7.8	14.8	32.4	28.9	36.6	18.0	13.0	12.7	14.0	5.9	17.9	10.3	16.9
math	8.0	5.5	10.3	10.1	12.5	10.7	11.6	13.0	7.1	8.2	5.9	3.7	9.4	8.9
bit shift	0.0	2.2	4.5	8.1	5.4	8.0	6.1	0.0	3.8	2.2	0.0	8.5	0.9	3.8
bit logic	0.0	0.0	0.0	14.2	11.0	17.9	0.3	0.0	1.9	3.6	0.0	5.7	0.0	4.2
Branches	12.0	10.9	15.5	4.0	5.8	2.3	5.1	17.4	10.5	16.0	13.6	14.7	19.2	11.3
Invoke	0.0	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.9	0.3	0.0	0.1
Others	0.0	1.0	0.6	0.2	2.5	2.4	9.4	0.0	7.1	4.7	2.2	4.2	5.9	3.1
Total	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
STACK (bytes)														
Max. stack	6	8	4	24	20	14	10	6	18	16	12	22	16	13.5
Avg. stack	2.08	2.37	2.14	11.76	6.30	6.77	3.36	1.89	2.73	3.15	2.19	4.83	3.08	4.1
	B.sort	H.sort	Bin.Search	XXTEA	MD5	RC5	FFT	Outlier	LEC	CoreMark	MoteTrack	HeatCalib	HeatDetect	average

Table 7.4: Performance data per benchmark

	B.sort	H.sort	Bin.Search	XXTEA	MD5	RC5	FFT	Outlier	LEC	CoreMark	MoteTrack	HeatCalib	HeatDetect	average
PERFORMANCE OVERHEAD USING ORIGINAL SOURCE (% of native C)														
Total	1277.1	1927.2	1319.4	714.5	470.6	409.9	437.8	549.0	885.3	809.7	1018.7	210.2	203.9	787.2
push/pop	640.1	356.7	233.7	197.2	115.7	70.1	66.6	207.2	106.6	220.4	166.5	80.9	78.8	195.4
load/store	360.1	197.4	175.3	67.0	46.7	33.2	29.3	190.3	110.7	136.8	218.2	67.6	43.8	129.0
mov(w)	10.0	41.1	8.4	6.6	3.6	0.1	5.2	21.5	5.1	5.5	38.6	-3.0	9.5	11.7
other	266.9	331.4	902.1	82.8	104.0	67.8	76.8	130.1	370.6	234.2	220.0	37.4	65.6	222.3
vm	0.0	1000.6	0.0	361.1	200.4	238.7	260.0	-0.1	292.2	212.9	375.4	27.3	6.2	228.8
OVERHEAD REDUCTION FROM SOURCE CODE OPTIMISATION (% of native C)														
Source optimisation	-613.2	-1234.0	-843.6	-464.1	-244.2	-285.6	-315.0	-56.5	-612.7	-433.7	-227.9	0.0	1.7	-409.9
PERFORMANCE OVERHEAD BEFORE COMPILER OPTIMISATIONS (% of native C)														
Total	663.9	693.2	475.8	250.4	226.4	124.3	122.8	492.5	272.6	376.0	790.8	210.2	205.6	377.3
push/pop	266.9	200.8	202.2	166.4	105.3	61.9	57.2	205.5	105.6	123.8	137.7	80.9	77.5	137.8
load/store	240.3	177.5	191.0	42.5	43.9	28.5	25.2	190.4	111.7	89.2	165.3	67.6	47.6	109.3
mov(w)	23.3	14.8	4.5	3.9	2.6	-1.2	4.2	8.0	5.1	5.3	17.6	-3.0	10.9	7.4
other	133.5	118.4	78.1	37.7	74.6	35.1	36.2	88.8	49.0	97.7	94.8	37.4	63.4	72.7
vm	0.0	181.7	0.0	0.0	0.0	0.0	0.0	-0.1	1.1	60.0	375.4	27.3	6.2	50.1
OVERHEAD REDUCTION PER COMPILER OPTIMISATION (% of native C)														
Impr. peephole	-233.5	-157.7	-149.4	-60.3	-48.2	-23.1	-36.5	-186.9	-54.2	-58.8	-60.2	-35.2	-54.5	-89.1
Stack caching	-40.0	-56.0	-57.3	-98.4	-58.0	-39.8	-16.2	-27.8	-67.7	-40.7	-63.1	-41.4	-24.2	-48.6
Pop. val. caching	-133.1	-84.9	-67.4	-6.8	-12.9	-8.8	-10.7	-51.0	-28.8	-24.5	-41.5	-15.4	-15.5	-38.5
Mark loops	-102.9	-46.8	-85.4	+5.0	-10.9	-8.0	-7.9	-114.9	-18.0	-40.0	-54.3	-38.2	-28.6	-42.4
Const shift	0.0	-17.1	-35.4	-18.4	-45.2	-20.9	-3.8	0.0	-9.6	-10.1	0.0	-17.2	-3.3	-13.9
16-bit array index	-53.2	-34.9	-15.7	-13.9	-5.5	-4.2	-2.8	-36.2	-9.7	-38.9	-19.7	-1.7	-9.0	-18.9
SIMUL	0.0	0.0	0.0	0.0	0.0	0.0	-27.2	0.0	0.0	-36.6	0.0	0.0	0.0	-4.9
Lightw. methods	0.0	-207.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-67.5	-395.7	-30.6	-0.3	-54.0
PERFORMANCE OVERHEAD AFTER COMPILER OPTIMISATIONS (% of native C)														
Total	101.2	88.5	65.2	57.6	45.7	19.5	17.7	75.7	84.6	58.9	156.3	30.5	70.2	67.0
push/pop	0.0	-2.8	0.0	37.4	0.1	2.9	2.0	-0.2	-13.7	2.5	20.4	5.6	1.7	4.3
load/store	1.0	29.3	27.0	-2.3	20.3	4.3	2.4	4.5	54.3	17.1	72.0	2.7	13.5	18.9
mov(w)	10.0	9.4	11.8	5.6	1.5	0.1	2.9	6.8	7.4	9.6	14.9	5.1	4.4	6.9
other	90.2	52.5	26.4	16.9	23.8	12.2	10.4	64.7	35.5	28.8	35.7	17.0	46.1	35.4
vm	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-0.1	1.1	0.8	13.2	0.0	4.4	1.5
	B.sort	H.sort	Bin.Search	XXTEA	MD5	RC5	FFT	Outlier	LEC	CoreMark	MoteTrack	HeatCalib	HeatDetect	average

7.3 AOT translation: performance

Next we will look at the effect of our optimisations to the baseline AOT translation approach, for all our benchmarks.

The trace data produced by Avrora gives us a detailed view into the run-time performance and the different types of overhead. We count the number of bytes and cycles spent on each native instruction for both the native C and our AOT compiled version, and group them into 4 categories that roughly match the types of AOT translation overhead discussed in Section 5.1.2:

- `PUSH`,`POP`: Matches the type 1 push/pop overhead since native code uses almost no push/pop instructions.

- `LD`,`LDD`,`ST`,`STD`: Matches the type 2 load/store overhead and directly shows the amount of memory traffic.

- `MOV`,`MOVW`: For moves the picture is less clear since the AOT compiler emits them for various reasons. Without stack caching, it emits moves to replace push/pop pairs, and after adding the mark loops optimisation to save a pinned value when it is popped destructively.

- others: the total overhead, minus the previous three categories. This roughly matches the type 3 overhead.

We define the overhead from each category as the number of bytes or cycles spent in the AOT version, minus the number spent in the native version for that category, and again normalise this to the *total* number of bytes or cycles spent in the native C version. The detailed results for each benchmark and type of overhead are shown in tables 7.4 and 7.5. In addition, Table 7.4 also lists the time spent in the VM on method calls and allocating objects. The constant array optimisation is already included in these results, since MoteTrack cannot run without it. We will examine its effect separately in section 7.6.

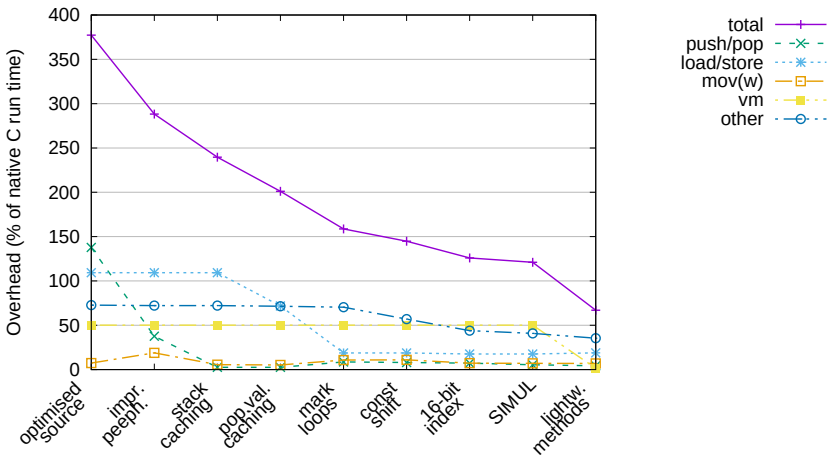

Figure 7.1: Performance overhead per category

The baseline shown in Table 7.4 before optimisation is the original, direct translation of the C code to Java. This results in a large overhead of up to 20x slowdown for *heap sort*, most of which is due to method calls since small functions and macros in the C code are not inlined in this version. Optimising the source code reduces overhead dramatically, but this is partly because the other optimisations, which target some of the same overhead, have not yet been applied. For example, in Table 7.4 optimising the source code reduces *CoreMark*'s overhead by 434%, while the previous section showed that when all other optimisations are applied first, the difference is only 268%. Since the source code optimisations were discussed in the previous section, the rest of this evaluation will focus on the effect of the other optimisations on the already optimised source.

Figure 7.1 starts with the manually optimised source code and incrementally adds each optimisation to the AOT compilation process to show how they combine to reduce performance overhead. We take the average of all benchmarks, and show both the total overhead, and the overhead for each instruction category. Figure 7.2 shows the total overhead for each individual benchmark.

Using the baseline AOT compilation on the optimised sources, the types 1, 2 and 3 overhead are all significant, at 138%, 109%, and 73% respectively, and the 50% overhead in the VM is mainly spent on method calls since the overhead from allocating temporary objects is removed by the source code optimisations. The basic approach does not have many reasons to emit a move, so in some cases the AOT version actually spends fewer

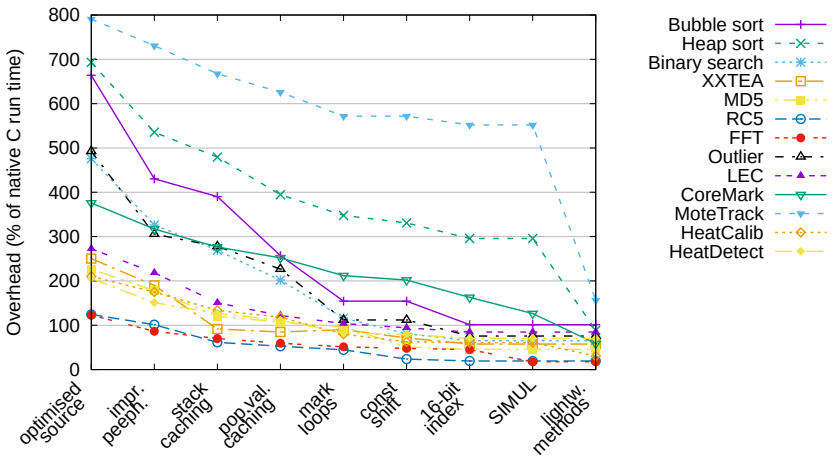

Figure 7.2: Performance overhead per benchmark

cycles on move instructions than the C version, resulting in small negative values. When we improve the peephole optimiser to include non-consecutive push/pop pairs, push/pop overhead drops by 100.2% (of native C performance), but if the push and pop target different registers, they are replaced by a move instruction, and we see an increase of 11.5% in move overhead. For a 16-bit register pair this takes 1 cycle (for a `MOVW` instruction), instead of 8 cycles for two pushes and two pops. The increase in moves shows most of the extra cases that are handled by the improved optimiser are replaced by a move instead of eliminated, since the 11.5% extra move overhead matches a 92% reduction in push/pop overhead.

Next stack caching is introduced to utilise all available registers and eliminate most of the push/pop instructions that cannot be handled by the improved optimiser. As a result the push/pop overhead drops to nearly 0, and so does the move overhead since most of the moves introduced by the peephole optimiser, are also unnecessary when using stack caching.

Having eliminated the type 1 overhead almost completely, popped value caching is added to remove a large number of the unnecessary load instructions. This reduces the memory traffic significantly, as is clear from the reduced load/store overhead, while the other types remain stable. Adding the mark loops optimisation further reduces loads, and this time also stores, by pinning common variables to a register. But it uses slightly more move instructions, and the fact that fewer registers are available for stack caching means

stack values are spilled to memory more often. While 53.0% is saved on loads and stores, the push/pop and move overhead increase by 6.0% and 5.6% respectively.

Most of the push/pop and load/store overhead has now been eliminated and the type 3 overhead, unaffected by these optimisations, has become the most significant source of overhead. This type has many different causes, part of it can be eliminated with the instruction set optimisations. These optimisations, especially the 16-bit array index, also reduce register pressure, which results in a slight decrease in the other overhead types, although this is minimal in comparison. The *CoreMark* and *FFT* benchmarks are the only ones to do 16-bit to 32-bit multiplication, so the average performance improvement for `SIMUL` is small, but Table 7.4 shows it is significant for these two benchmarks.

Finally, the lightweight optimisation could be applied to almost every method. Lightweight methods still incur some overhead, which will be discussed in more detail in Section 7.7, but since they do not call the VM, the time spent in the VM on method calls is effectively eliminated.

Combined, the optimisations to the AOT compilation process reduce performance overhead from 377% to 67% of native C performance.

Table 7.5: Code size data per benchmark

	B.sort	H.sort	Bin.Search	XXTEA	MD5	RC5	FFT	Outlier	LEC	CoreMark	MoteTrack	HeatCalib	HeatDetect	average
CODE SIZE OVERHEAD USING ORIGINAL SOURCE (% of native C)														
Total	449.2	298.0	208.2	287.2	166.0	239.3	94.9	316.3	186.4	159.4	255.0	26.2	238.5	225.0
push/pop	159.3	99.3	71.2	140.6	110.7	108.6	47.7	92.6	60.7	69.6	78.1	31.7	93.9	89.5
load/store	128.8	65.8	76.7	68.9	40.8	56.5	20.3	103.2	71.4	51.6	75.9	22.6	56.4	64.5
mov(w)	1.7	17.4	9.6	10.1	-3.6	0.0	2.5	14.7	5.7	-3.1	24.1	-14.3	15.1	6.1
other	159.3	115.4	50.7	67.6	18.0	74.3	24.5	105.8	48.6	41.2	76.9	-13.8	73.1	64.7
OVERHEAD REDUCTION FROM SOURCE CODE OPTIMISATION (% of native C)														
Source optimisation	-195.0	-58.4	-26.0	-124.2	+45.9	+110.2	+4.5	-47.4	+4.3	-31.2	-27.7	0.0	-12.7	-27.5
CODE SIZE OVERHEAD BEFORE COMPILER OPTIMISATIONS (% of native C)														
Total	254.2	239.6	182.2	163.0	211.9	349.5	99.4	268.9	190.7	128.2	227.3	26.2	225.8	197.5
push/pop	71.2	80.5	60.3	103.7	133.3	165.3	52.6	86.3	63.6	55.2	72.8	31.7	83.3	81.5
load/store	88.1	73.8	74.0	28.4	56.7	67.9	19.7	101.1	72.9	45.8	68.2	22.6	60.1	59.9
mov(w)	10.2	9.4	4.1	2.6	-1.0	2.2	4.3	4.7	5.7	-3.4	19.6	-14.3	16.2	4.6
other	84.7	75.8	43.8	28.2	22.9	114.1	22.8	76.8	48.6	30.5	66.7	-13.8	66.2	51.3
OVERHEAD REDUCTION PER COMPILER OPTIMISATION (% of native C)														
Impr. peephole	-67.8	-53.0	-45.2	-38.3	-49.4	-62.5	-32.2	-77.8	-33.9	-24.7	-27.4	-13.6	-49.8	-44.3
Stack caching	-25.4	-26.2	-24.7	-59.4	-85.4	-111.2	-20.9	-30.6	-39.7	-27.6	-26.7	-12.6	-38.3	-40.7
Pop. val. caching	-16.9	-29.5	-6.8	-6.2	-18.7	-18.7	-13.5	-5.2	-18.5	-9.9	-26.7	-8.1	-20.7	-15.3
Mark loops	+1.7	0.0	+21.9	+5.9	-1.2	-2.6	-4.2	-16.4	+2.5	-1.5	-8.7	-1.3	-11.4	-1.2
Const shift	0.0	-6.1	-6.9	+1.7	+2.8	-16.0	-4.6	-2.6	-1.8	-1.5	0.0	-1.7	-0.1	-2.8
16-bit array index	-27.2	-22.8	-8.2	-11.6	-5.1	-16.7	-11.6	-25.8	-10.7	-7.4	-16.9	-2.2	-10.7	-13.6
SIMUL	0.0	0.0	0.0	0.0	0.0	0.0	-9.9	0.0	0.0	-3.4	0.0	0.0	0.0	-1.1
Lightw. methods	0.0	-2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-5.5	-3.8	-3.9	+0.6	-1.1
CODE SIZE OVERHEAD AFTER COMPILER OPTIMISATIONS (% of native C)														
Total	118.6	100.0	112.3	55.1	54.9	121.8	2.5	110.5	88.6	46.7	117.1	-17.2	95.4	77.4
push/pop	23.7	16.1	27.4	13.3	0.0	6.2	1.9	-2.1	-5.0	1.7	16.3	3.9	-3.1	7.7
load/store	33.9	41.6	49.3	14.8	37.2	25.3	-2.6	57.9	45.0	30.1	40.9	8.0	37.6	32.2
mov(w)	1.7	6.7	6.8	2.5	-2.4	11.9	-0.8	1.1	7.1	-0.2	15.4	-10.7	13.3	4.0
other	59.3	35.6	28.8	24.4	20.1	78.5	4.0	53.7	41.4	15.1	44.5	-18.4	47.6	33.4
	B.sort	H.sort	Bin.Search	XXTEA	MD5	RC5	FFT	Outlier	LEC	CoreMark	MoteTrack	HeatCalib	HeatDetect	average

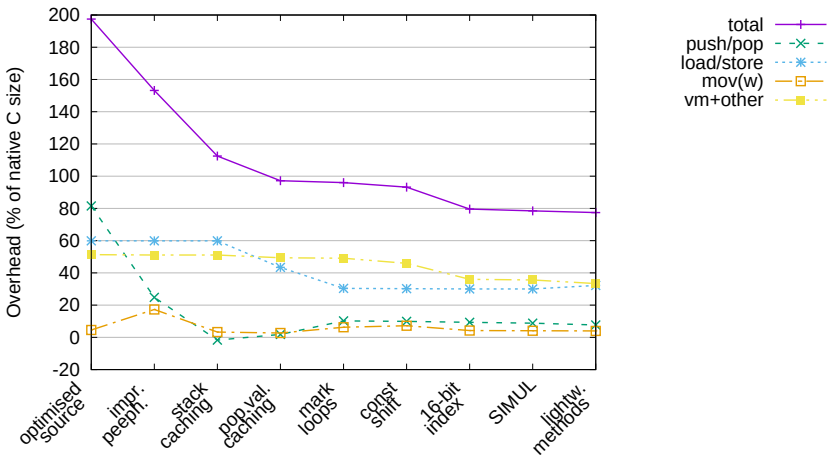

Figure 7.3: Code size overhead per category

7.4 AOT translation: code size

Next we examine the effects of our optimisations on code size. Two factors are important here: the size of the VM itself and the size of the code it generates.

The size overhead for the generated code is shown in figures 7.3 and 7.4, again split up per instruction category and benchmark respectively. For the first three optimisations, the two graphs follow a similar pattern as the performance graphs. These optimisations eliminate the need to emit certain instructions, which reduces code size and improves performance at the same time.

The mark loops optimisation moves loads and stores for pinned variables outside of the loop. This reduces performance overhead by 42%, but the effect on code size varies per benchmark: some are slightly smaller, others slightly larger.

For each variable that is live at the beginning of the loop, the VM emits the load before the mark loop block, so code size is only reduced if the variable is loaded more than once. Code size may actually increase if the value is then popped destructively, since this causes the VM to emit a `mov`. Stores follow a similar argument. Also, for small methods the extra registers used may mean more call-saved registers have to be saved in the method prologue. Finally, we get the performance advantage for each run-time iteration, but the effect on code size, whether positive or negative, only once.

The constant shift optimisation unrolls the loop that is normally generated for bit shifts.

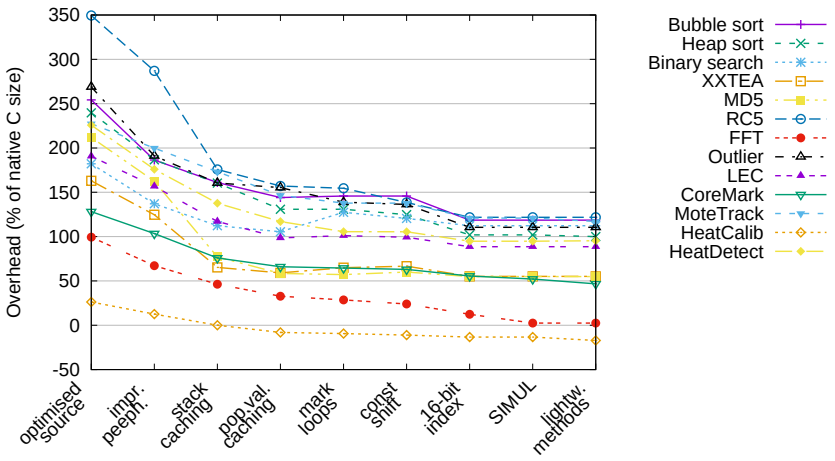

Figure 7.4: Code size overhead per benchmark

This significantly improves performance, but the effect on the code size depends on the number of bits to shift by. The constant load and loop take at least 5 instructions. In most cases the unrolled shifts are smaller, but *MD5* and *XXTEA* show a small increase in code size since they contains shifts by a large number of bits.

Using 16-bit array indexes also reduces code size. The benchmarks here already have the manual source code optimisations, so they use short index variables. This means the infuser emits `S2I` instructions to cast them to 32-bit ints if the array access instructions expect an int index. Not having to emit those when the array access instructions expect a 16-bit index, and the reduced work the access instruction needs to do, saves 14% code size overhead in addition to the 19% reduction in performance overhead. Using 32-bit variables in the source code also removes the need for `S2I` instructions, but the extra effort needed to manipulate the 32-bit index variable would make the net code size even larger.

7.4.1 VM code size and break-even point

These more complex code generation techniques do increase the size of our compiler. The first column in Table 7.6 shows the difference in code size between the AOT translator and Darjeeling's interpreter. The basic AOT approach is 6863 B larger than the interpreter, and each optimisation adds a little to the size of the VM.

They also generate significantly smaller code. The third column shows the reduction

in the generated code size compared to the baseline approach. Here we show the reduction in total size, as opposed to the overhead used elsewhere, to be able to calculate the break-even point. Using the improved peephole optimiser adds 276 bytes to the VM, but it reduces the size of the generated code by 14.6%. If we have more than 1.9 KB available to store user programmes, this reduction will outweigh the increase in VM size. Adding more complex optimisations further increases the VM size, but compared to the baseline approach, the break-even point is well within the range of memory typically available on a sensor node, peaking at at most 17.8 KB.

As is often the case, there is a trade-off between size and performance. The interpreter is smaller than each version of our AOT compiler, and Table 7.5 shows bytecode is smaller than both native C and AOT compiled code, but the interpreter's performance penalty may be unacceptable in many cases. Using AOT compilation we can achieve adequate performance, but the most important drawback has been an increase in generated code size. These optimisations help to mitigate this drawback, and both improve performance, and allow us to load more code on a node.

For the smallest devices, or if we want to be able to load especially large programmes, we may decide to use only a selection of optimisations to limit the VM size and still get both a reasonable performance, and most of the code size reduction. For example, dropping the markloop optimisation would reduces the size of the VM by 3 KB but keeps most of the reduction in generated code size, while performance overhead would increase to around 109%.

7.4.2 VM memory consumption

The last column in Table 7.6 shows the amount of data that needs to be kept in memory while translating a method. We would like our VM to be able to load new code while other tasks are running concurrently. Here we only list the data that the VM would need to maintain in between receiving messages with new code, since this is the amount of memory that would not be available to other tasks during this process. Of course, when new code is being processed, more stack memory is used, but this is freed after a batch of

Table 7.6: Code size and memory consumption

	Size vs interpreter	Size vs baseline		AOT code reduction	Break even	Memory usage
Baseline	6863 B					25 B
Improved peephole	7139 B	276 B	(+276)	-14.6%	1.9 KB	25 B
Simple stack caching	7961 B	1098 B	(+822)	-27.8%	3.9 KB	36 B
Popped value caching	9229 B	2366 B	(+1268)	-33.1%	7.1 KB	80 B
Markloop	12511 B	5648 B	(+3282)	-33.4%	16.9 KB	87 B
Const shift	12955 B	6092 B	(+444)	-34.3%	17.8 KB	87 B
16-bit array index	12935 B	6072 B	(-20)	-38.7%	15.7 KB	87 B
SIMUL	13001 B	6138 B	(+66)	-39.2%	15.7 KB	87 B
Lightweight methods	13549 B	6686 B	(+548)	-39.7%	16.8 KB	87 B

The constant shift optimisation adds 170 bytes to the VM size. Because the MoteTrack benchmark cannot run without it, we cannot calculate the average code size reduction.

instructions has been processed and can be reused by other applications.

For the baseline approach we only use 25 bytes for a number of commonly used values such as a pointer to the next instruction to be compiled, the number of instructions in the method, etc. The simple stack caching approach adds a 11 byte array to store the state of each register pair we use for stack caching. Popped value caching adds two more arrays of 16-bit elements to store the value tag and age of each value. Mark loops only needs an extra 16-bit variable to mark which registers are pinned, and a few other variables. Finally, the instruction set optimisations do not require any additional memory. In total, our compiler requires 87 bytes of memory during the compilation process.

7.5 Benchmark details

Next, we have a closer look at some of the benchmarks and see how the effectiveness of each optimisation depends on the characteristics of the source code. The first section of Table 7.4 shows the distribution of the bytecode instructions executed in each benchmark, and both the maximum and average number of bytes on the VM stack. We can see some important differences between the benchmarks. While the sort benchmarks on the left are almost completely load/store bounded, *XXTEA*, *RC5* and *MD5* are much more computation intensive, spending fewer instructions on loads and stores, and more on math or bitwise operations. The left three benchmarks and the *outlier detection* benchmark have

only a few bytes on the stack, but as the benchmarks contain more complex expressions, the number of values on the stack increases.

The second part of tables 7.4 and 7.5 first shows the overhead before optimisation, split up in the five instruction categories. We then list the effect of each optimisation on the total overhead. Finally we show the overhead per category after applying all optimisations.

The improved peephole optimiser and stack caching both target the push/pop overhead. Stack caching can eliminate almost all, and replaces the need for a peephole optimiser, but it is interesting to compare the two. The improved peephole optimiser does well for the simple benchmarks like sorting, *binary* search and *outlier detection*, leaving less overhead to remove for stack caching. The more computation intensive benchmarks contain more complicated expressions, which means there is more distance between a push and a pop, leaving more cases that cannot be handled by the peephole optimiser. For these benchmarks, replacing the peephole optimiser with stack caching yields a big improvement.

The benchmarks on the left spend more time on load/store instructions. This results in higher load/store overhead, and the two optimisations that target this overhead, popped value caching and mark loops, have a big impact. For the computation intensive benchmarks, the load/store overhead is much smaller, but the higher stack size means stack caching is very important for these benchmarks.

The smaller benchmarks highlight certain specific aspects of our approach, while the larger *CoreMark* benchmark covers a mix of different types of processing. As a result, it is an average case in almost every row in Table 7.4.

Bubble sort

Next we look at *bubble sort* in some more detail. After optimisation, most of the stack related overhead has been eliminated and of the 101.2% remaining performance overhead, most is due to other sources. For *bubble sort* there is a single, clearly identifiable source. The detailed trace output shows that 79.8% is due to `ADD` instructions, but *bubble sort* hardly does any additions. This is a good example of how the simple JVM instruction set leads to less efficient code. To access an array we need to calculate the address of

the indexed value, which takes one move and five additions for an array of shorts. This calculation is repeated for each access, while the C version is more efficient, using the auto-increment version of the AVR's LD and ST instructions to slide a pointer over the array. Of the remaining 101.2% overhead, 93.1% is caused by these address calculations.

HeatCalib and FFT

Table 7.5 shows that after optimisation the *HeatCalib* benchmark has a negative code size overhead. This is caused by the fact that we compile the C versions using `avr-gcc`'s -O3 optimisations, optimising for performance instead of code size. In this case, as well as for *FFT*, this caused `avr-gcc` to duplicate a part of the code, which improves performance but at the cost of a significantly larger code size.

MoteTrack

The *MoteTrack* benchmark is by far the slowest of our benchmarks, at a 156% overhead compared to native C. *MoteTrack* stores a database of reference signatures in flash memory. In C this is a complex struct containing a number of sub-structures and fixed-sized arrays, while it becomes a collection of objects and arrays in Java, shown in Figure 8.1.

Since the layout of the complete C structure is known at compile time, the C function to load a reference signature from its database can simply use `memcpy_P` to copy a block of 80 bytes from flash memory to RAM. For Java, the method to read from flash memory must follow multiple pointers to follow several indirections to find the locations to put each value. As a result, reading a single signature takes 1455 cycles in Java, and only 735 cycles in C.

Additionally, the fixed offsets in the C structure means that accessing the reference signatures is also more efficient in C than in Java, which must follow a number of references to reach the data. We discuss this in more detail in Section 8.3.

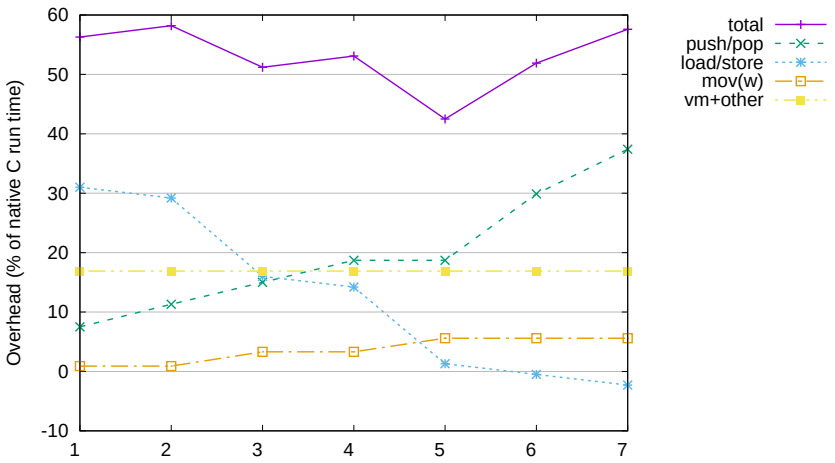

Figure 7.5: XXTEA performance overhead for different number of pinned register pairs

LEC

In Section 1.2.1 we calculated that the LEC compression algorithm reduced the energy spent to transmit our sample ECG data by 650 µJ, at the expense of 246 µJ spent on CPU cycles compressing the data, when implemented in C and using the ATmega128 CPU and CC2420 radio.

A compression algorithm like LEC is a good example of an optimisation that may be part of an application loaded onto a sensor node. However, if the overhead of using a VM is too high, the cost of compression may outweight the energy saved on transmission. Table 7.4 shows that using the baseline AOT approach, the *LEC* benchmark has an overhead of 272.6%. This means the CPU has to stay active longer, and compressing the data would cost $246\mu J * 3.726 \approx 916\mu J$, which is more than the 650 µJ saved on transmission.

After we apply our optimisations, the overhead is reduced to 84.6%, resulting in $246\mu J * 1.846 \approx 454\mu J$ spent on compression. While the savings are less than when using native C to compress the data, our optimisations mean that in this scenario, we can save on transmission costs by using LEC compression, while using the baseline AOT approach, LEC compression would have resulted in a net loss.

XXTEA and the mark loops optimisation

The *XXTEA* benchmark has the highest average stack depth of all benchmarks. As a result, popped value caching does not have much effect: most registers are used for real stack

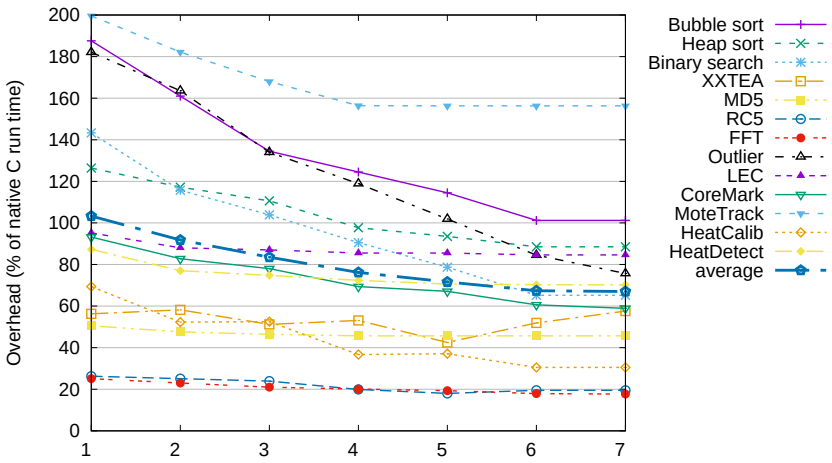

Figure 7.6: Per benchmark performance overhead different number of pinned register pairs

values, leaving few chances to reuse a value that was previously popped from the stack.

When we apply the mark loops optimisation, performance actually degrades by 5%! Here we have an interesting trade-off: if we use a register to pin a variable, accessing that variable will be cheaper, but this register will no longer be available for stack caching, so more stack values may have to be spilled to memory.

For most benchmarks, using the maximum of 7 register pairs to pin variables was also the best option. At a lower average stack depth, the fewer number of registers available for stack caching is easily compensated for by cheaper variable access. For *XXTEA* however, the cost of spilling more stack values to memory outweighs the gains of pinning more variables when too many variables are pinned. Figure 7.5 shows the overhead for *XXTEA* from the different instruction categories. When we increase the number of register pairs used to pin variables from 1 to 7, the load/store overhead steadily decreases, but the push/pop and move overhead increase. For *XXTEA*, the optimum is at 5 pinned register pairs, at which the total overhead is only 43%, instead of 58% at 7 pinned register pairs.

Interestingly, when we pin 7 pairs, the AOT version does fewer loads and stores than the C compiler. Under high register pressure the C version may spill a register value to memory and later load it again, adding extra load/store instructions. When the AOT version pins too many registers, it will also need to spill values, but this adds push/pop instructions instead of loads/stores.

Table 7.7: 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%	17.7%	17.7%	86.5%	84.6%	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

Figure 7.6 shows the performance for each benchmark, as the number of pinned register pairs is increased. The benchmarks stay stable or even slow down when the number pinned pairs is increased beyond 5 are the benchmarks that have a high stack depth, while the benchmarks with low stack depth such as sort, *binary search* and *outlier detection* improve significantly. It should be possible to develop a simple heuristic to allow the VM to make a better decision on the number of registers to pin. Since our current VM always pins 7 pairs, we used this as our end result and leave this heuristic to future work.

7.6 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 7.7. 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.

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 that will create an array object on the heap and fill each element individually, as shown in Listing 4.

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 930 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.

7.7 Method invocation

In this section we will examine the effect of the lightweight method, compared to inlined code and normal method calls.

Most of our smaller benchmarks consist of only a single method. ProGuard automatically inlines methods only called from a single location, eliminating all method calls in the *LEC* benchmark. We will examine the effect of lightweight methods using the *FFT*, *Heap sort*, and *CoreMark* benchmarks.

FFT contains a single function, `fix_mpy`, which was inlined by the C compiler, so we manually inlined it in the Java version. *Heap sort* contains a real method call: it consist of two loops, both repeatedly calling the `siftDown` method. Since it is called from two places, and is larger than ProGuard's size threshold, it is not automatically inlined. *CoreMark* is a much more extensive benchmark, and consists of many methods, only four of which could be inlined by ProGuard.

Table 7.8: Methods per benchmark and relative performance for normal, lightweight invocation, and inlining

	# calls	C	Java Base version	Java Alternative version	Java Using normal method calls
CoreMark					
ee_isdigit	3920	normal (inlined)	manually inlined	lightweight (handw.)	manually inlined
core_state_transition	1024	normal	lightweight	lightweight	normal
crcu8	584	normal (inlined)	manually inlined	manually inlined	manually inlined
crcu16	292	normal	lightweight	lightweight	normal
calc_func	223	normal	lightweight	lightweight	normal
compare_idx	209	normal (inlined)	Proguard inlined	Proguard inlined	Proguard inlined
core_list_find	206	normal	lightweight	lightweight	normal
compare_complex	110	normal	Proguard inlined	Proguard inlined	Proguard inlined
crcu32	64	normal	lightweight	lightweight	normal
matrix_sum	16	normal	lightweight	lightweight	normal
others (<16 calls each)	39	normal	normal	normal	normal
cycles			2,705,654	2,863,302	3,855,242
overhead v native C			58.9%	68.2%	126.4%
code size			8990	9006	9,328
FFT					
FIX_MPY	768	marked inline	manually inlined	lightweight(handw.)	normal
cycles			179,692	215,020	661,360
overhead v native C			17.7%	40.8%	333.1%
code size			1,342	1,320	1,408
heap sort					
SWAP	1642	#define	manually inlined	manually inlined	manually inlined
siftDown	383	normal	lightweight	manually inlined	normal
cycles			208,239	184,071	437,264
overhead v native C			88.5%	66.6%	295.8%
code size			596	662	602

Highlights indicate changes from the versions used to obtain the results in the previous sections.

Table 7.8 lists the functions of the *CoreMark*, *FFT* and *heap sort* benchmarks, and the number of times they are called in a single run. Next, we list the way they are implemented in C. *CoreMark* only defines normal functions, which are inlined by `avr-gcc` in three cases. *FFT*'s `fix_mpy` function is marked with the `inline` compiler hint, which was followed by `avr-gcc`. Finally *heap sort* uses just one extra function, and a macro to swap two array elements.

The Java base version column shows the way these functions are implemented in the Java versions of our benchmarks. We manually inlined C macros, and the functions that were inlined by the C compiler. The most commonly called methods were transformed to lightweight methods, simply by adding the `@Lightweight` annotation.

In the next two columns we vary these choices slightly to examine the effect of lightweight

methods.

For the *CoreMark* benchmark, we first replace the inlined implementation of the most frequently called method with a lightweight version. `ee_isdigit` returns true if a `char` passed to it is between `'0'` and `'9'`. Since this is a very trivial method, we manually wrote the bytecode for this lightweight method to use only the stack and no local variables. This slowed down the benchmark by 6%, adding 157,648 cycles. Since the method is called 3920 times, this corresponds to an overhead of about 40 cycles, which is on the high side for such a small method.

This is due to another overhead from using a lightweight method that's hard to quantify: the boolean result of `ee_isdigit` is used to decide an `if` statement. When we inline the code, the VM can directly branch on the result of the expression `(c>='0' && c<='9')`, but the lightweight method first has to return a boolean, which is then tested after the lightweight call returns.

Next, we see what the performance would be without lightweight methods, and all methods, except the manually inlined `ee_isdigit` and `crcu8`, have to be implemented as normal Java methods. This adds a total of 1,149,588 cycles, making it 1.42 times slower than the lightweight methods version. Spread over 1,825 calls, this means the average method invocation added over 630 cycles, which is within the range predicted in Section 5.4.

The *FFT* benchmark has a much lower running time than *CoreMark,* but still does 768 function calls. In the C and normal Java versions these are inlined. When we change `FIX_MPY` to a normal Java method, it is too large for ProGuard to inline. Using a hand-written lightweight method, the large number of calls relative to the total running time means the average overhead of over 46 cycles per invocation slows down the benchmark by 20%. Without lightweight methods, the overhead would be 627 per call, slowing down the benchmark by 268%.

Finally, for the *heap sort* benchmark we normally use a lightweight method for `siftDown`. While manually inlining it is possible, it is not an attractive option since the `siftDown` method is much more complex than `FIX_MPY` or `ee_isdigit`. When we do inline it,

we see that the lightweight method adds some overhead compared to the inlined version, but much less than a normal method call would.

In terms of code size, using normal methods take slightly more space than a lightweight method. Listing 6 showed that the invocation is more complex for normal methods, and in addition the method prologue and epilogue are longer.

The difference between inlining and lightweight methods is less clear. For the smallest of methods, such as *CoreMark*'s `ee_isdigit`, the inlined code is slightly smaller than the call, but the *heap sort* benchmark shows that inlining larger methods can result in significantly larger code. Since `siftDown` is called from two places, duplicating it leads to a 11% increase for the 16-bit version of *heap sort*. For the 32-bit version, where `siftDown` is relatively larger, this increases to 27%.

As these three examples show, using lightweight methods gives us an option in between a normal method call and inlining. This avoids most of the overhead of a normal method call, and the potential size increase of inlining.

Table 7.9: Cost of safety guarantees

	B.sort	H.sort	Bin.Search	XXTEA	MD5	RC5	FFT	Outlier	LEC	CoreMark	MoteTrack	HeatCalib	HeatDetect	average
EXECUTED BYTECODE INSTRUCTIONS (% of total executed bytecode instructions after optimisation)														
Array element/object field STORES	18.0	7.8	0.0	2.9	4.5	1.5	6.1	5.8	3.7	2.6	10.0	1.4	4.7	5.3
Array element/object field LOADS	18.0	15.9	7.1	8.6	6.2	6.4	7.0	10.7	7.9	11.7	21.4	4.1	8.8	10.3
PERFORMANCE OVERHEAD VS NATIVE C (% of native C)														
unsafe	101.2	88.5	65.2	57.6	45.7	19.5	17.7	75.7	84.6	58.9	156.3	30.5	70.2	67.0
safe writes	247.5	153.9	65.2	68.2	60.3	22.2	30.3	128.4	118.4	76.7	266.1	33.9	88.2	104.6
safe reads and writes	393.9	287.8	151.7	100.0	80.3	33.4	43.0	226.6	179.8	155.0	445.1	43.9	120.8	173.9
PERFORMANCE OVERHEAD VS UNSAFE VM (% of unsafe AOT)														
safe writes	72.7	34.7	0.0	6.7	10.0	2.3	10.7	30.0	18.3	11.2	42.8	2.6	10.6	22.5
safe reads and writes	145.5	105.7	52.4	26.9	23.7	11.6	21.5	85.9	51.6	60.5	112.7	10.3	29.7	64.0
CODE SIZE OVERHEAD VS NATIVE C (% of native C)														
unsafe	118.6	100.0	112.3	55.1	54.9	121.8	2.5	110.5	88.6	46.7	117.1	-17.2	95.4	77.4
safe writes	125.4	105.4	112.3	56.2	55.7	125.3	5.0	118.9	94.3	50.5	125.4	-16.4	102.6	81.6
safe reads and writes	132.2	113.4	117.8	60.1	59.1	132.3	8.0	123.2	102.9	58.2	145.3	-13.9	106.2	88.1
CODE SIZE OVERHEAD VS UNSAFE VM (% of unsafe AOT)														
safe writes	3.1	2.7	0.0	0.7	0.5	1.6	2.4	4.0	3.0	2.6	3.8	1.0	3.7	2.4
safe reads and writes	6.2	6.7	2.6	3.2	2.7	4.7	5.4	6.0	7.6	7.8	13.0	4.0	5.5	6.0

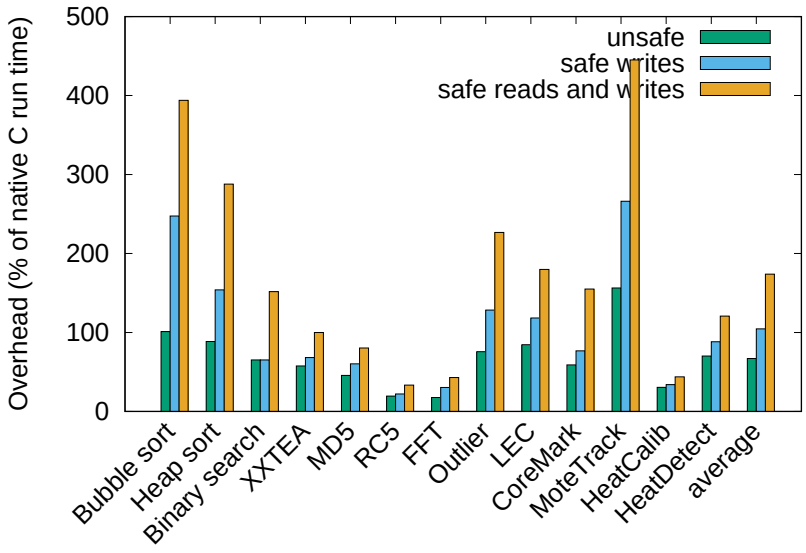

Figure 7.7: Overhead increase due to safety checks

7.8 The cost of safety

The advantage of using a VM to provide safety is that the necessary checks are easy to do, compared to native code, and most can be done at translation time. This leads to both a very modest increase in VM complexity due to the safety checks, and a lower run-time overhead.

7.8.1 Run-time cost

Table 7.9 shows the increase in performance and code size overhead as a result of the run-time safety checks for our 12 benchmarks. The performance overhead is also shown in Figure 7.7. The baseline here is the unsafe version of our VM, which is on average 67.0% slower than native C. Checking heap *writes* is sufficient to satisfy our guarantee that no malicious code can corrupt the state of the VM. This increases the average overhead to 104.6% of native C, corresponding to a 22.5% increase in run time compared to the unsafe VM.

The cost of the run-time safety checks depends greatly on the benchmark we run. Most checks are done at translation time, including writes to local and static variables. The only check that adds significant run-time overhead is check R-4, which checks the target of an

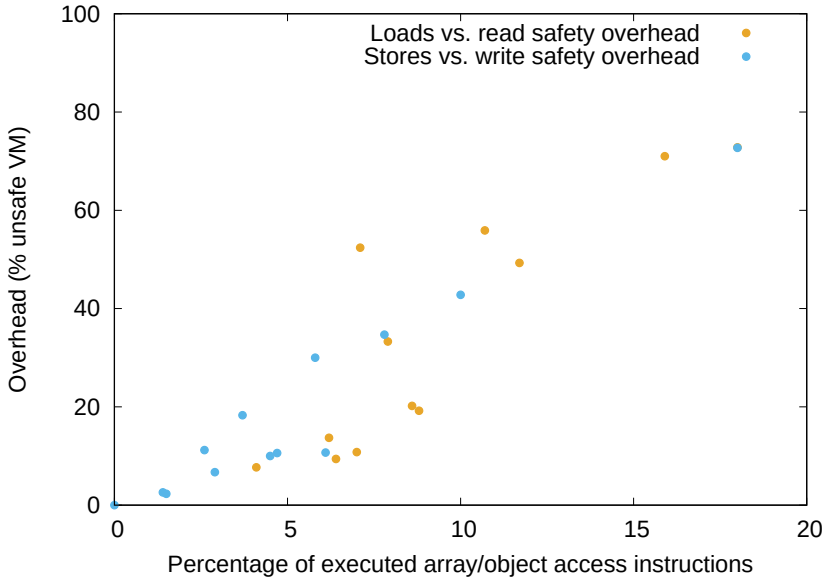

Figure 7.8: Percentage of array/object load/store instructions and cost of read/write safety

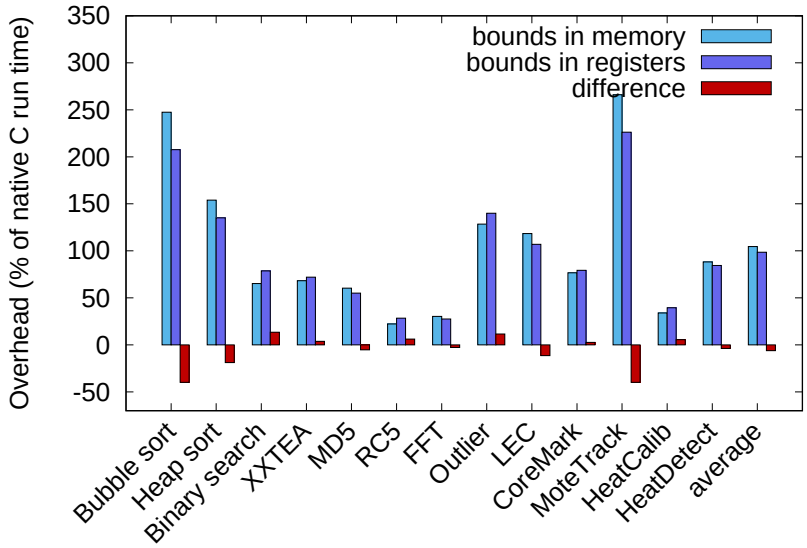

Figure 7.9: Comparison of safety cost with heap bounds in memory or registers

object field or array write is within the bounds of the heap.

Thus, the run-time overhead is determined by the number of object or array accesses a benchmark does. The percentage of these is shown in the first part of Table 7.9. Since *bubble sort* has by far the highest percentage of array writes, at 18% of all executed bytecode instructions, it also incurs the highest overhead from adding write safety, and slows down by 72.7%. *Binary search* on the other hand, which does no writes at all, is unaffected. As usual *CoreMark*, being a large benchmark with a mix of operations, is somewhere in the middle. The correlation between the percentage of array and object writes, and the slowdown compared to the unsafe version is shown in Figure 7.8.

Safe reads

Up to this point the VM only checks the application cannot *write* to memory it is not supposed to write to, however, it may still *read* from any location.

The recently published Meltdown and Spectre vulnerabilities in desktop CPUs can be exploited by malicious code to read from anywhere in memory, exposing both kernel's and other applications private data, which may contain sensitive information such as authentication tokens, passwords, etc. This sent OS vendors rushing to release patches, which early report suggest may cause a performance penalty of up to 11% [81].

Whether this is also a problem on a sensor node depends on the scenario. If the VM or other tasks contain sensitive information, then this may need to be protected. However, in many sensor node applications the node may only be running a single application, and CapeVM does not contain any state that would be useful to an attacker. In these cases, write safety will be sufficient.

Adding read safety to our VM is trivial: instructions to load local and static variables are already protected since they use the same code to access a variable as the store instructions. For heap access, we simply add the same call to `heapcheck` to the `GETARRAY` and `GETFIELD` instructions just before the actual read.

Figure 7.7 shows the cost of providing read safety is higher than write safety. Most applications read from an array or object much more frequently than they write to them.

As a result, our VM with read and write safety turned on slows down by 64% on average, corresponding to a 174% slowdown over native C. In addition to the sort benchmarks, *MoteTrack* also suffers greatly from adding read safety, since it spends 21% of its instructions reading from the objects and arrays, mostly from the RSSI signature database. *RC5* is the fastest benchmark, since it not only does relatively few array reads and writes, but also spends a large amount of time on expensive variable bit shifts, which have identical performance in both C and AOT compiled versions. The result is a slowdown of only 33% compared to native C for the fully safe version.

Keeping heap bounds in registers

In Section 6.2.4 several alternatives for the heap bounds check were considered, one of which is to keep the bounds in dedicated registers to avoid having to fetch them from memory for each check. Here we evaluate this choice.

Having the bounds in registers would reduce the cost of the check from 22 to 14 cycles, reducing the overhead of safety checks by $8/22 \approx 36\%$. However, this uses 4 registers which we cannot use for stack caching.

To estimate how this would affect performance, the benchmarks were run using the unsafe VM, with the number of registers available to the stack cache reduced by 4. Since this does not affect the number of heap accesses, we then added the observed overhead for safety checks, reduced by 36%.

Figure 7.9 shows the overhead for our chosen approach with the heap bounds in memory, compared to the expected overhead when the heap bounds are stored in registers. For some benchmarks such as *bubble sort* and *MoteTrack*, the savings in heap bounds checks outweighs the reduced effectiveness of the stack cache. But the improvement in performance is relatively small, and for other benchmarks the reverse is true, showing minor slowdowns when heap bounds are kept in registers. On average the benchmarks are quite balanced, as is the larger *CoreMark* benchmark.

As future work we may consider using some basic statistics, such as the percentage of array write instructions and average stack depth, to choose one of the two options on a

per-method basis. But as usual there is a trade-off, in this case VM size and complexity, which may not be worth the effort given the relatively small gains.

7.8.2 Code-size cost

Next, we examine the cost of safety in terms of code size. This comes in two parts: increased VM complexity and size, and an increase in the code it generates.

Most of our checks are no more complex than comparing two integers, and rejecting or terminating the application if a condition is not met. The most complex part is deciding the stack effects of instructions to guard against stack under- or overflow. This comes in the form of a table that encodes the effects of most instructions, and some specialised code to analyse a handful of instructions without fixed effect. In total, the increase in VM size for our safe version is a modest 1,468 bytes.

As we can see in Table 7.9, the size of the code the VM generates increases by only 2.4% for write only safety and 6.0% when reads are also protected. Since many checks occur at translation time, most instructions produce exactly the same native code in the safe version of our VM. The exceptions are `INVOKEVIRTUAL` and `INVOKEINTERFACE`, which now contain the expected stack effects to realise check R-3, and the array and object write instructions `PUTFIELD` and `PUTARRAY`, that emit a single extra `CALL` instruction the `heapcheck` routine. Since these instructions are both relatively rare, and already generate a larger than average block of native instructions, the total effect on code size is limited.

7.8.3 Comparison to native code alternatives

As discussed in Section 3.7, several non-VM approaches have been proposed to guarantee safety on a sensor node. Two of these, *t-kernel* [35] and Harbor [49], allow the node to guarantee safety independent of the host. Both target the Mica family of sensor nodes, which use the same ATmega128 CPU used by CapeVM. In this section we compare them to CapeVM and consider the question whether a VM is a good way to provide safety.

t-kernel reports a slowdown of between 50 and 200%, which is roughly in the same

Table 7.10: Comparison of overhead in Harbor and CapeVM

Benchmark	CapeVM overhead			Harbor overhead	
	VM	+ safety checks	= safe VM	current	hypothetical
Array writes	182.8%	268.7%	451.5%	1230%	416%
Outlier	75.7%	52.7%	128.4%	690%	234%
16-bit FFT	17.7%	12.6%	30.3%	380%	129%

range as CapeVM. However both *t-kernel* and CapeVM provide additional advantages. In *t-kernel*'s case a form of virtual memory, and for the VM platform independence. This makes them hard to compare, but we note that while the performance of both systems is similar, *t-kernel*'s code size overhead is much higher at a 6-8.5x increase, limiting the size of programmes that can be loaded onto the device.

A better comparison is possible for Harbor, which only provides safety. Harbor uses three benchmarks to evaluate performance: writing arbitrary data to an array to mimick copying a buffer of sensor data, and the *outlier detection*, and 16-bit *FFT* benchmarks also used in the rest of CapeVM's evaluation. The size of the data used for the first two is not specified in the paper, but since it mentions they work on sensor data and the ATmega CPU has 10-bit analog-to-digital converters, we use 16-bit data for these benchmarks.

As mentioned before, the *FFT* benchmark is taken from the Harbor sources [93], and *outlier detection* implemented as described in the paper. The *array writes* benchmark is implemented as a loop that fills an array of 256 elements with an arbitrary number, as shown in Listing 11.

```
1     for (short i = 0; i < NUMBERS; i++) {
2         numbers[i] = (short)1;
3     }
```

Listing 11: Array writes benchmark (8-bit version)

The resulting overhead is shown in Table 7.10. Filling an array is a hard case for safe CapeVM since consecutive array writes are expensive for two reasons: (i) it results in repeated executions of the `PUTARRAY` instruction, which calculates the target address for each write, while native code can slide a pointer over the array, and (ii) each of these writes will trigger a call to `heapcheck`.

CapeVM incurs overhead both related to the VM, and because of the added run-time safety checks, while for Harbor all overhead is due to safety checks. Still, CapeVM's total overhead of 451.5% is much lower than Harbor's 1230%.

While CapeVM is more than twice as fast as Harbor for this benchmark, the comparison is not entirely fair. Harbor lists the cycle overhead for all of its 5 run-time protection primitives. We assume that without any function calls, only the 'Write access check' is relevant to this benchmark, which takes 65 cycles. In contrast, CapeVM's `heapcheck` routine only takes 22 cycles.

The difference is due to Harbor's more fine grained protection, which allows it to grant access to any aligned block of 8 bytes to the application, while CapeVM's protection of the entire heap as a single block is more coarse. If Harbor could be modified to use a similar check, its overhead could potentially be reduced to $1230/65 * 22 \approx 416\%$, slightly faster than CapeVM's total overhead. However, it is not clear from the paper whether Harbor's architecture could support such a coarse-grained check since it requires all application data that needs run-time write checks to be in a single block of memory.

While this shows CapeVM achieves a performance comparable to even a hypothetical optimised version of Harbor, the *array writes* benchmark does not highlight the advantage of using a VM to provide safety because it spends much of its time writing to an array, for which both approaches insert a run-time check. However, the VM can verify writes to local and static variables to be safe at translation time, while the Harbor sources [93] show that its verifier requires *all* stores to go through the run-time write access check. The authors do note that static analysis of the code could reduce the number of checks, but that this would come at the cost of a significantly more complex verifier.

CapeVM's *total* overhead for the *array writes* benchmark is slightly higher than the hypothetical optimised Harbor, but the overhead due to safety checks is lower because CapeVM does not need to check writes to the index variable `i`. This advantage should be more pronounced in code with more frequent writes to local variables, which is exactly the case for the more realistic *outlier detection* and *FFT* benchmarks.

The reported overhead is 690% and 380%, which would result in 234% and 129%

Table 7.11: Number or registers and word size for the ATmega, MSP430, and Cortex M0

	ATmega [65, 64]	MSP430 [87, 86]	Cortex M0 [4]
Number of general purpose registers	32	12	13
Word size	8-bit	16-bit	32-bit
Total register file size (bytes)	32	24	52

resp., when using the faster memory access check. For these benchmarks, CapeVM is significantly faster at only 128.5% and 30.3% overhead.

7.9 Expected perfomance on other platforms

Platform independence is one of the main advantages of using a VM. The ATmega family of CPUs is widely used in low power embedded systems, and we implemented our VM for the popular ATmega128 CPU. However, our approach does not depend on any ATmega specific properties and the approach described in this dissertation can be applied on other embedded CPU platforms to improve performance and provide a safe execution environment. The main requirements are the ability to reprogramme its own programme memory, and the availability of a sufficient number of registers for stack caching.

While it is impossible to determine exactly what the resulting performance would be on different platforms without porting the VM, we present some results here that indicate it is likely to be slightly worse than the results we see on the ATmega.

Two important parameters that influence performance are the number of available registers and the size of the registers. Table 7.11 lists these parameters for the ATmega, and two other common families of embedded CPUs, the Texas Instruments MSP430 and ARM Cortex-M0. These CPUs are similar in many ways, including the amount of RAM and flash memory typically available, but differ in the number of registers and word size.

7.9.1 Number of registers

The ATmega has 32 8-bit registers available, which are managed as 16 pairs since the VM stores data in 16-bit slots. Looking at the MSP430, it only has 12 registers, but they

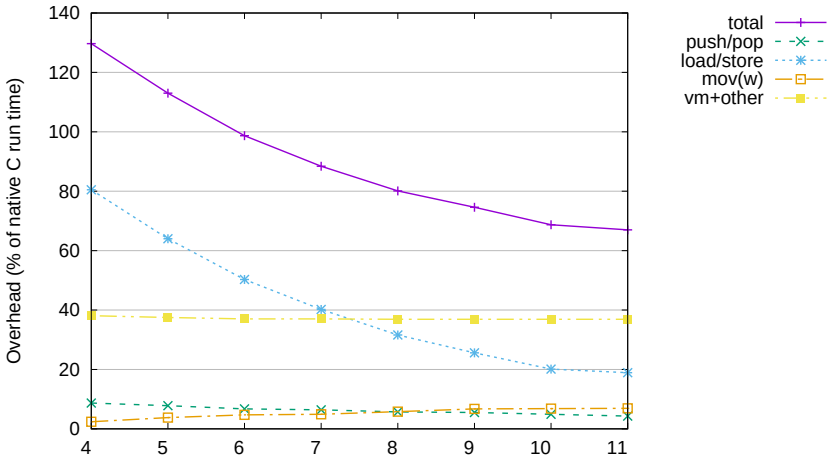

Figure 7.10: Performance for different stack cache sizes (in pairs of registers)

Table 7.12: Performance for different stack cache sizes (in pairs of registers)

Number of register pairs	Overhead push/pop	load/store	mov(w)	vm+other	total
4	8.7	80.5	2.4	38.1	129.7
5	7.8	64.0	3.8	37.5	113.0
6	6.7	50.3	4.7	37.0	98.7
7	6.4	40.2	4.9	37.0	88.4
8	5.7	31.6	5.8	36.9	80.1
9	5.5	25.6	6.7	36.9	74.6
10	4.9	20.1	6.8	36.9	68.7
11	4.3	18.9	6.9	36.9	67.0

are 16-bit. 5 register pairs are reserved on the ATmega, leaving 11 pairs available for stack caching. Assuming the same can be achieved by reserving 5 16-bit registers on the MSP430, 7 registers will be available to the stack cache.

To evaluate the effect of a smaller number of registers on the stack cache, all benchmarks were run while restricting the number of registers the cache manage may use. Since the VM needs a minimum of 4 pairs to implement all instructions, we vary the number of register from 4 to 11.

The results are shown in Figure 7.10 and Table 7.12. As is common with caching techniques, the first few registers have the most impact, with half of the overhead reduction already realised when adding the first two additional registers. Ignoring all other difference for the moment, the effect of reducing the number of register pairs from 11 to 7 is an increase in overhead by 21.4%, to 88.4%.

The Cortex M0 has one general purpose register more than the MSP430, and at 8

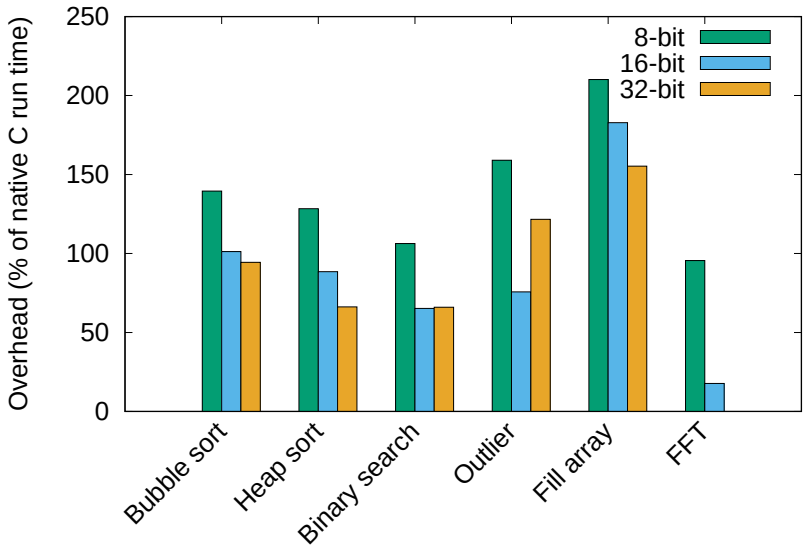

Figure 7.11: Performance for different data sizes

Table 7.13: Performance for different data sizes

	8-bit	16-bit	32-bit
Bubble sort	139.5	101.2	94.4
Heap sort	128.4	88.5	66.2
Binary search	106.3	65.2	66.0
Outlier	159.0	75.7	121.6
Fill array	210.1	182.8	155.3
FFT	95.5	17.7	

register pairs the overhead drops to 80.1%. In addition, the M0's registers are 32-bit, which means in cases where mostly 32-bit values are used, the cache size is effectively doubled since values can be stored in a single register instead of two pairs of 8-bit registers.

A final interesting thing to note in Figure 7.10 is the fact that increasing the cache size mostly reduces the load/store overhead. Using only 4 pairs, stack caching has already removed most of the push/pop overhead, which only drops slightly when more registers are used. However, these extra registers reduce load/store overhead significantly since more registers are available for the markloop optimisation, and using more registers increases the chance that an old, popped value may still be present, allowing popped value caching to eliminate more loads.

7.9.2 Word size

A second important difference between the CPUs in Table 7.11 is the size of the registers. The main measure to evaluate our approach has been the overhead compared to native C performance or code size. While having a large register size is good for absolute performance, we expect it to hurt the *relative* performance of our VM.

A number of benchmarks can be implemented using different data sizes. As mentioned in Section 7.1, 16-bit data is used in the main evaluation. Figure 7.11 and Table 7.13 show the resulting performance for 8, 16, and 32-bit versions of these benchmarks (no 32-bit version of `fix_fft.c` was available).

In almost all cases, a smaller data size results in a higher relative overhead. This is because this reduces the time spent in code that is common to both C and AOT versions. For example, the different versions usually do the same number of memory accesses. When operating on 16-bit data, this only takes half as long as for 32-bit data, while the surrounding overhead either stays the same or is reduced by a smaller factor, thus increasing the relative overhead.

Taking *bubble sort* as an example, both versions do the same number of loads, comparisons, and array stores. *bubble sort*'s overhead primarily comes from calculating array element locations (see Section 7.5). 32-bit array access takes 17 cycles, 8 of which are spent on the actual load or store. For 16-bit array access this is only 4 out of 11 cycles, which drops further to 2 out of 10 cycles for 8-bit arrays. Thus the relative overhead for accessing smaller elements is higher.

This effect is even clearer for more complex operations like multiplication. 16x16 to 16-bit multiplication can be implemented in a few instructions and only takes 10 cycles on the ATmega. 32x32 to 32-bit multiplication is implemented by calling `avr-gcc`'s `__mulsi3` function, and takes 85 to 100 cycles.

Thus, working with 16-bit or 32-bit data on an 8-bit CPU helps the VM's relative performance by introducing a larger common component that both C and AOT compiled versions have to execute. On the MSP430 or Cortex M0, that operate on 16-bit or 32-bit values in a single step, this effect will be reduced or eliminated. The exact impact of this

is hard to estimate, but it is likely to be in the order of tens of percents extra overhead.

In Table 7.13 we see the *outlier detection* and *binary search* benchmarks perform worse for 32-bit data compared to 16-bit data. This is due to a mismatch between the infuser and ProGuard. In JVM bytecode, local variables are stored in 32-bit slots. The code generated by `javac` uses a separate slot for each variable, but ProGuard attempts to reduce memory consumption by mapping multiple variables to the same slot if their live ranges do not overlap. The infuser processes the ProGuard optimised code, as shown in Figure 4.1, and replaces the JVM's 32-bit operations by 16-bit versions where possible. In the 32-bit *outlier detection* benchmark, ProGuard mapped a 32-bit and 16-bit variable to the same slot, which prevents the infuser from using the cheaper 16-bit operations for this variable. This once again highlights the need for a unified, optimising compiler, combining the tasks of `javac`, ProGuard and the infuser.

The large difference in performance for *FFT* is mostly due to bit shifts. The 8-bit version does many shifts of a 16-bit value by exactly 6 bits. The VM simply emits 6 single-bit shifts, while `avr-gcc` has a special optimised version for this case, which we considered too specific to include in the VM. The 32-bit version spends about half of its time shifting a 32-bit value by 15 bits. The VM and `avr-gcc` both implement this using the same loop, which again adds a large common factor, thus reducing the relative overhead.

7.10 Limitations and the cost of using a VM

The quantitative evaluation in the previous sections has shown that AOT compilation techniques can reduce the performance overhead of using a VM to within a range that will be acceptable for many applications. However, this is not the only cost associated with using a VM. This section discusses the limitations of CapeVM, and the cost of using a VM compared to native code.

Since CapeVM is based on Darjeeling, we share many of its limitations. Like Darjeeling, CapeVM does not support multidimensional arrays, reflection, constant data, 64-bit or floating point data types [13]. In addition, CapeVM drops support for exceptions and

threads since they are much harder to implement in an AOT compiler than in an interpreter. As we will argue in Chapter 8, we feel that if the goal is to provide useful, platform independent and safe reprogramming of sensor node with adequate performance, instead of simply porting Java to a sensor node, many of Java's more advanced features, especially those that are expensive to implement, should be replaced by more lightweight alternatives.

Besides these unsupported features, there are other costs to using CapeVM when compared to native code. One of the most important concerns is size. While our optimisations reduce the code size overhead significantly, AOT compiled code is still larger than native C and the VM itself also takes up space. In terms of RAM, the heap adds a 5 byte overhead to each object or array, and we have seen that Java cannot represent complex structures like *MoteTrack*'s RSSI signature efficiently. For code that only uses a limited number of large objects or arrays like the *heat detection* benchmark, this overhead will be acceptable, but for code using many tiny objects like *MoteTrack* this overhead is significant.

In terms of performance, a limitation of lightweight methods is that they don't support recursive function calls. However, we have not found such code in any benchmark, and the limited amount of RAM on a sensor node means recursion would be a bad choice in most cases.

When optimising code for performance, small choices can often have unexpected consequences. We found this to be much more significant when writing Java code for our VM, than when writing the same algorithms in C where `avr-gcc`'s optimisations often mean two different approaches in C result in similar binary code. An optimising combined compiler and infuser will help a Java developer by doing some of the same optimisations, but there are many examples where the most natural way to solve a problem in Java may not result in the best performance. For example, Suganuma [84] notes that Java code typically results in many small methods and invocations, which can result in a serious performance penalty if they cannot be made lightweight or eliminated by automatic inlining. Allocating many temporary objects hurts performance, while creating them once and reusing them usually results in slightly unnatural code. While this is also true for C, the impact of C

function calls and allocating temporary objects on the stack is much smaller.

The next chapter will look at some of these limitations and propose ideas on how to improve them in future sensor node VMs.

Chapter 8

Lessons from JVM

In Section 1.4 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. In this chapter we consider the third question: whether Java is a suitable language for a sensor node VM, and how it may be improved.

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 [38]. 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 standard Java.

In this chapter we discuss the most pressing issues we encountered, summarised in Table 8.1, and suggest ways they could be improved in future VMs. Where possible, the impact of these issues is quantified in Table 8.2.

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 [30] is a sensor node version of C, but some

Table 8.1: Point requiring attention in future sensor node VMs

Section	Issue	in	affects
8.1	A tailored standard library	Standard library	VM size
8.2	Support for constant arrays	Source language, VM	memory usage, code size
8.3	Support for nested data structures	Source language, VM	memory usage, performance
8.4	Better lang. support for shorts and bytes	Source language	memory usage, source maintainability
8.5	Simple type definitions	Source language	source maintainability
8.6	Explicit and efficient inlining	Source language	performance
8.7	An optimising compiler	Compiler	performance
8.8	Allocating objects on stack	Source language, VM	(predictable) performance
8.9	Reconsidering adv. language features threads, exceptions, OO, garbage collection	Source language, VM	VM size, complexity, and performance

require more drastic changes.

8.1 A tailored standard library

A minimum Java API for resource-constrained devices, the Connected Limited Device Configuration (CLDC) specification, was proposed by Sun Microsystems [69]. 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 8.3 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 [7] 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 8.2: 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
8.2	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		
8.3	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	1996	1387 [c]	676	1158
	Size increase	1.6%	1.6%	4.0%	11.1%	23.0%	12.5%	6.3%	2.8%	3.3%	22.2%	128.9%	5.0%	6.4%
8.4	Casts	1	6	5	8	8	8	16	3	10	70	33	4	64
	Lines of code [d]	11	24	16	38	165	27	73	44	77	849	475	51	266
	Casts per 100 LOC	9	25	31	21	5	30	22	7	13	8	7	8	24
8.6	Slowdown non-inlined version		69%		57%	25%	37%	20%			13%			
	Size difference non-inlined version		+42		-224	-1502	-94	-20			+48			
8.7	Slowdown w/o optimisations	91%	52%	544%	3%			3%	23%		117%	76%		2%
8.8	Slowdown from heap allocations									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 8.3: 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

8.2 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 5.3.5.

8.3 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 12 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 12: 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 8.1. As described in Section 2.2.2, 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 12 to Java, is as a collection of objects and arrays on the heap, as shown in the right half of Figure 8.1. Note that every one of the 18 `RFSignal` structs becomes an object, which in turn has a pointer to an array of RSSI values.

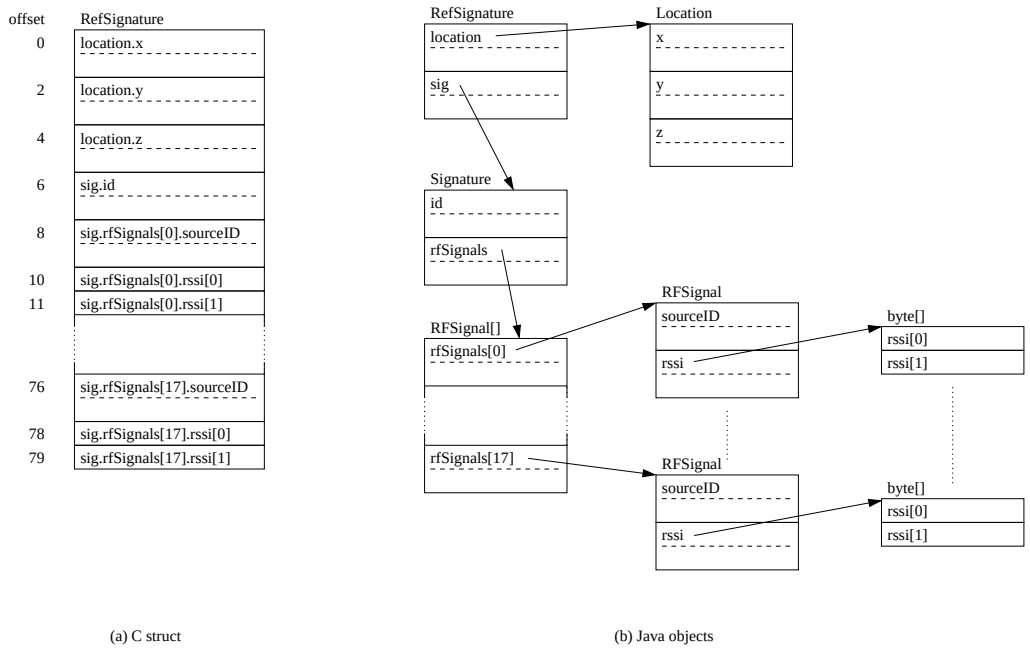

Figure 8.1: The `RefSignature` data structure: as a C struct, and as a collection of Java objects

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, 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 variables stored directly in `RFSignal`, as explained in Section 7.1.1. 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.

Table 8.2 shows the size of the main data structures used by each benchmark. For most benchmarks that operate on a handful of objects and arrays, the overhead is limited to at most tens of bytes. Beside's *MoteTrack*, the *CoreMark* benchmark also has a significant overhead but here the cause is the linked list of `ListHead` and `ListData` objects, each of which has a 5 byte header.

Generally, large arrays of primitive types do not suffer from this problem and can be stored with low relative overhead, but for programmes containing large numbers of small objects the overhead is significant. In some cases we can work around the problem by flattening the structure. Section 7.2 described an alternative for *CoreMark* to replace the linked list with a single array, and in *MoteTrack*'s case we could replace the array of `RFSignal` objects with three separate arrays for `sourceID`s, `rssi_0` and `rssi_1`. In both cases the reduced memory overhead comes at a significant cost in readability.

8.4 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 store the sum of b and c in a, 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 small 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 8.2 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`.

8.5 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.

8.6 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 implementing it as a lightweight method slows down the benchmark by 57%. Tools like Proguard [36] can be used to inline small methods or methods that are only called from a single location. But in this case, `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.

Inlining larger methods called from multiple locations does run the risk of increasing code size. However, the data in Table 8.2 shows the increase in code size is often modest. In two cases the inlined version is actually smaller. The code generated by the AOT compiler for a lightweight method call is still larger than for most other bytecode instructions, and in *CoreMark*'s case the inlining the `ee_isdigit` method means the code directly branches on the result of the inlined expression, instead of having to perform an additional branch on the returned boolean value.

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.

8.7 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. Table 8.2 shows the resulting performance without the manual optimisations discussed in Section 5.2. These could be further expanded, and combining the tasks of the optimiser and infuser can further improve performance as shown in Section 7.9.2.

8.8 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* 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.

This overhead can often be reduced by allocating earlier and reusing the same objects in a loop. In Listing 13 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

```
    public static short LEC(short[] numbers, Stream stream) {
        BSI bsi = new BSI();       // Allocate bsi only once
        for (...) {
            ...
            compress(ri, ri_1, stream, bsi);
            ...
        }
    }

    private static void compress(short ri, short ri_1, Stream stream, BSI bsi) {
        ...
        encode(di, bsi);           // Pass bsi to encode to return both value and length
        ...
    }

    private static void encode(short di, BSI bsi) {
        ...
        bsi.value = ...            // return value and length by setting object fields
        bsi.length = ...
    }
```

Listing 13: Avoiding multiple object allocations in the LEC benchmark

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 three benchmarks mentioned before. Table 8.2 shows the performance without this optimisation. 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 [17, 32] 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 [7]. 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.

8.9 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 8.3 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 8.1 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 to implement 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.

8.9.1 Threads

As shown in Table 8.3, 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* [34], nesC [30], and more recently Amulet [40].

8.9.2 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.

8.9.3 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.

8.9.4 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 8.3 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.

8.10 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, and the data in Table 8.2, suggest 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 9

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 1.4, 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 61%, resulting in an average performance overhead of 67%, and the code size overhead of 77% 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 23% 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 CapeVM to existing work on improving sensor node VM performance, and on safe execution environments in Table 9.1.

Table 9.1: Comparison of CapeVM to related work

Approach	Platform indep.	Safe	Performance	Code size
Native code	No	No	1x	1x
Interpreters	Yes	Mostly no	300-55400% slower	50% smaller [b]
Ellul's AOT	Yes	No	123-810% slower	26-350% larger [b]
Safe TinyOS	No	Yes [a]	17% slower	27% larger
Harbor	No	Yes	380 to 1230% slower	30 to 65% larger
t-kernel	No	Yes	50 to 200% slower	500 to 750% larger
CapeVM (unsafe)	Yes	No	67% slower	77% larger
CapeVM (safe)	Yes	Yes	105% slower	82% 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
```

```
        byte bits_left_current_in_byte = (byte)(8 - stream.bits_used_in_current_byte);
        while (bsi_length > 0) {
            if (bsi_length > bits_left_current_in_byte) {
                // Not enough space to store all bits

                // Calculate bits to write to current byte
                byte bits_to_add_to_current_byte =
                        (byte)(bsi >> (bsi_length - bits_left_current_in_byte));

                // Add them to the current byte
                stream.data[stream.current_byte_index] |= bits_to_add_to_current_byte;
                // Remove those bits from the to-do list
                bsi_length -= bits_left_current_in_byte;

                // Advance the stream to the next byte
                stream.current_byte_index++;
                // Whole new byte for the next round
                bits_left_current_in_byte = 8;
            } else {
                // Enough space to store all bits

                // After this we'll have -bsi_length bits left.
                bits_left_current_in_byte -= bsi_length;

                // Calculate bits to write to current byte
                byte bits_to_add_to_current_byte =
                        (byte)(bsi << bits_left_current_in_byte);

                // Add them to the current byte
                stream.data[stream.current_byte_index] |= bits_to_add_to_current_byte;
                // Remove those bits from the to-do list
                bsi_length = 0;
            }
        }

        stream.bits_used_in_current_byte = (byte)(8 - bits_left_current_in_byte);
        // Note that if we filled the last byte, stream_bits_used_in_current_byte
        // will be 8, which means in the next call to encode the first iteration of
        // the while loop won't do anything, except advance the stream pointer.
    }

    // pseudo code:
    // encode(di)
    //     // compute di category
    //     IF di = 0
    //         SET ni to 0
    //     ELSE
    //         SET ni to CEIL(log_2(|di|))
    //     ENDIF
    //     // extract si from Table
    //     SET si TO Table[ni]
    //     // build bsi
    //     IF ni = 0 THEN
    //         // ai is not needed
    //         SET bsi to si
    //     ELSE
    //         // build ai
    //         IF di > 0 THEN
    //             SET ai TO (di)|ni
    //         ELSE
    //             SET ai TO (di-1)|ni
    //         ENDIF
    //         // build bsi
    //         SET bsi TO <<si,ai>>
    //     ENDIF
    //     RETURN bsi
    private static void encode(short di, BSI bsi) {
        // compute di category
        short di_abs;
        if (di < 0) {
            di_abs = (short)-di;
        } else {
            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 14: 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 15: 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_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 16: 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 consideredd 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;
    }
```

```
160
161     private static void get_filtered_xy(int[] largestSubset, int maxSubsetLen) {
162         short x_weight_zscore_sum = 0;
163         short y_weight_zscore_sum = 0;
164         int zscore_sum = 0;
165         byte low_zscore_length = 0;
166
167         short xh_weight_zscore_sum = 0;
168         short yh_weight_zscore_sum = 0;
169         int zscore_sum_h = 0;
170         byte hot_zscore_length = 0;
171
172         for(short i=0; i<maxSubsetLen; i++) {
173             short _zscore = HeatCalib.zscore[largestSubset[i]];
174             if (_zscore > zscore_threshold_hot) {
175                 zscore_sum_h += _zscore;
176                 hot_zscore_length += 1;
177             } else if (_zscore > zscore_threshold_low) {
178                 zscore_sum += _zscore;
179                 low_zscore_length += 1;
180             }
181         }
182
183         for(short i=0; i<maxSubsetLen; i++) {
184             int _x = largestSubset[i] % 8;
185             int _y = largestSubset[i] >>> 3;
186             short _zscore = HeatCalib.zscore[largestSubset[i]];
187             if (_zscore > zscore_threshold_hot) {
188                 xh_weight_zscore_sum += _x * _zscore / zscore_sum_h;
189                 yh_weight_zscore_sum += _y * _zscore / zscore_sum_h;
190             } else if (_zscore > zscore_threshold_low) {
191                 x_weight_zscore_sum += _x * _zscore / zscore_sum;
192                 y_weight_zscore_sum += _y * _zscore / zscore_sum;
193             }
194         }
195
196         if (hot_zscore_length > 0) {
197             xh_weight_coordinate = xh_weight_zscore_sum;
198             yh_weight_coordinate = yh_weight_zscore_sum;
199         }
200
201         if (low_zscore_length > 0) {
202             x_weight_coordinate = x_weight_zscore_sum;
203             y_weight_coordinate = y_weight_zscore_sum;
204         }
205     }
206
207     private static void get_xy(int[] largestSubset, int maxSubsetLen) {
208         short x_weight_zscore_sum = 0;
209         short y_weight_zscore_sum = 0;
210         int zscore_sum = 0;
211
212         for(short i=0; i<maxSubsetLen; i++) {
213             short _zscore = HeatCalib.zscore[largestSubset[i]];
214             zscore_sum += _zscore;
215         }
216
217         for(short i=0; i<maxSubsetLen; i++) {
218             int _x = largestSubset[i] % 8;
219             int _y = largestSubset[i] >>> 3;
220             short _zscore = HeatCalib.zscore[largestSubset[i]];
221             x_weight_zscore_sum += 100 * _x * _zscore / zscore_sum;
222             y_weight_zscore_sum += 100 * _y * _zscore / zscore_sum;
223         }
224         x_weight_coordinate = x_weight_zscore_sum;
225         y_weight_coordinate = y_weight_zscore_sum;
226     }
227
228
229     private static void labelPixel(int[] largestSubset, int maxSubsetLen, byte[] color)
230     {
231         for(short i=0; i<maxSubsetLen; i++) {
232             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
    {
```

```
303             for (short i=0; i<8; i++) {
304                 neighbor[i] = -1;
305             }
306             if((loc >>> 3)==0 && (loc % ARRAY_SIZE)==0){
307                 neighbor[0]=(short)(loc+1); neighbor[1]=(short)(loc+ARRAY_SIZE);
    ↪   neighbor[2]=(short)(loc+ARRAY_SIZE+1);
308             }else if((loc >>> 3)==0 && (loc % ARRAY_SIZE)>0 && (loc %
    ↪   ARRAY_SIZE)<(ARRAY_SIZE-1)){
309                 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);
310             }else if((loc >>> 3)==0 && (loc % ARRAY_SIZE)==(ARRAY_SIZE-1)){
311                 neighbor[0]=(short)(loc-1); neighbor[1]=(short)(loc+ARRAY_SIZE-1);
    ↪   neighbor[2]=(short)(loc+ARRAY_SIZE);
312             }else if((loc >>> 3)> 0 && (loc >>> 3)<(ARRAY_SIZE-1) && (loc % ARRAY_SIZE)==0){
313                 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);
314             }else if((loc >>> 3)>0 && (loc >>> 3)<(ARRAY_SIZE-1) && (loc % ARRAY_SIZE)>0 &&
    ↪   (loc % ARRAY_SIZE)<(ARRAY_SIZE-1)){
315                 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);
316             }else if((loc >>> 3)>0 && (loc >>> 3)<(ARRAY_SIZE-1) && (loc %
    ↪   ARRAY_SIZE)==(ARRAY_SIZE-1)){
317                 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);
318             }else if((loc >>> 3)==(ARRAY_SIZE-1) && (loc % ARRAY_SIZE)==0){
319                 neighbor[0]=(short)(loc-ARRAY_SIZE); neighbor[1]=(short)(loc-ARRAY_SIZE+1);
    ↪   neighbor[2]=(short)(loc+1);
320             }else if((loc >>> 3)==(ARRAY_SIZE-1) && (loc % ARRAY_SIZE)>0 && (loc %
    ↪   ARRAY_SIZE)<(ARRAY_SIZE-1)){
321                 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);
322             }else if((loc >>> 3)==(ARRAY_SIZE-1) && (loc % ARRAY_SIZE)==(ARRAY_SIZE-1)){
323                 neighbor[0]=(short)(loc-ARRAY_SIZE-1); neighbor[1]=(short)(loc-ARRAY_SIZE);
    ↪   neighbor[2]=(short)(loc-1);
324             }
325         }
326
327         private static void get_four_neighbor(short loc, short[] neighbor) //neighbor length
    ↪    maximum is 4
328         {
329             for (short i=0; i<4; i++) {
330                 neighbor[i] = -1;
331             }
332             if((loc >>> 3)==0 && (loc % ARRAY_SIZE)==0){
333                 neighbor[0]=(short)(loc+1); neighbor[1]=(short)(loc+ARRAY_SIZE);
334             }else if((loc >>> 3)==0 && (loc % ARRAY_SIZE)>0 && (loc %
    ↪   ARRAY_SIZE)<(ARRAY_SIZE-1)){
335                 neighbor[0]=(short)(loc-1); neighbor[1]=(short)(loc+1);
    ↪   neighbor[2]=(short)(loc+ARRAY_SIZE);
336             }else if((loc >>> 3)==0 && (loc % ARRAY_SIZE)==(ARRAY_SIZE-1)){
337                 neighbor[0]=(short)(loc-1); neighbor[1]=(short)(loc+ARRAY_SIZE);
338             }else if((loc >>> 3)> 0 && (loc >>> 3)<(ARRAY_SIZE-1) && (loc % ARRAY_SIZE)==0){
339                 neighbor[0]=(short)(loc-ARRAY_SIZE); neighbor[1]=(short)(loc+1);
    ↪   neighbor[2]=(short)(loc+ARRAY_SIZE);
340             }else if((loc >>> 3)>0 && (loc >>> 3)<(ARRAY_SIZE-1) && (loc % ARRAY_SIZE)>0 &&
    ↪   (loc % ARRAY_SIZE)<(ARRAY_SIZE-1)){
341                 neighbor[0]=(short)(loc-ARRAY_SIZE); neighbor[1]=(short)(loc-1);
    ↪   neighbor[2]=(short)(loc+1); neighbor[3]=(short)(loc+ARRAY_SIZE);
342             }else if((loc >>> 3)>0 && (loc >>> 3)<(ARRAY_SIZE-1) && (loc %
    ↪   ARRAY_SIZE)==(ARRAY_SIZE-1)){
343                 neighbor[0]=(short)(loc-ARRAY_SIZE); neighbor[1]=(short)(loc-1);
    ↪   neighbor[2]=(short)(loc+ARRAY_SIZE);
344             }else if((loc >>> 3)==(ARRAY_SIZE-1) && (loc % ARRAY_SIZE)==0){
345                 neighbor[0]=(short)(loc-ARRAY_SIZE); neighbor[1]=(short)(loc+1);
346             }else if((loc >>> 3)==(ARRAY_SIZE-1) && (loc % ARRAY_SIZE)>0 && (loc %
    ↪   ARRAY_SIZE)<(ARRAY_SIZE-1)){
347                 neighbor[0]=(short)(loc-ARRAY_SIZE); neighbor[1]=(short)(loc-1);
    ↪   neighbor[2]=(short)(loc+1);
```

```
348             }else if((loc >>> 3)==(ARRAY_SIZE-1) && (loc % ARRAY_SIZE)==(ARRAY_SIZE-1)){
349                 neighbor[0]=(short)(loc-ARRAY_SIZE); neighbor[1]=(short)(loc-1);
350             }
351         }
352 
353         private static boolean label_neighbor(int[] result, int subsetNumber){
354             boolean hasNeighbor = false;
355             for(short i=0; i<64; i++){
356                 if(result[i]==subsetNumber){
357                     get_eight_neighbor(i, neighbor);
358                     for(short j=0; j<8;j++){
359                         if(neighbor[j] != -1 && result[neighbor[j]] == WAITTOCHECK){
360                             result[neighbor[j]]=NEIGHBOR;
361                             hasNeighbor = true;
362                         }
363                     }
364                 }
365             }
366             return hasNeighbor;
367         }
368 
369         private static boolean check_neighbor_zscore_weight(short index) {
370             boolean rv = false;  // done this way to avoid values on the stack at a brtarget
371             get_four_neighbor(index, neighbor);
372             for(short i=0; i<4;i++){
373                 if (neighbor[i] != -1) {
374                     if (zscoreWeight[neighbor[i]]) {
375                         rv = true;
376                         break;
377                     }
378                 }
379             }
380             return rv;
381         }
382     }
383 
384     public class ShortWrapper {
385         public short value;
386     }
```

Listing 17: Heat detection benchmark source code (detection phase)

Bibliography

[1] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. *OOPSLA '99: Proceedings of the Fourteenth ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Nov. 1999.

[2] Anon. Python-on-a-chip. `http://code.google.com/p/python-on-a-chip`, 2011.

[3] Anon. Heap sort at Rosetta Code. `https://rosettacode.org/mw/index.php?title=Sorting_algorithms/Heapsort&oldid=214878#C`, 2015.

[4] ARM. Cortex M0 Technical Reference Manual Revision r0p0. 2009.

[5] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y. Choi, T. Herman, S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita. A Line in the Sand: A Wireless Sensor Network for Target Detection, Classification, and Tracking. *Elsevier Computer Networks*, 46(5), Dec. 2004.

[6] F. Aslam. TakaTuka source code. `http://sourceforge.net/p/takatuka`.

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

[8] F. Aslam, C. Schindelhauer, G. Ernst, D. Spyra, J. Meyer, and M. Zalloom. Poster Abstract: Introducing TakaTuka - A Java Virtual Machine for Motes. *SenSys '08: Proceedings of the Sixth International Conference on Embedded Networked Sensor Systems*, Nov. 2008.

[9] R. Balani, C.-C. Han, R. K. Rengaswamy, I. Tsigkogiannis, and M. Srivastava. Multi-level Software Reconfiguration for Sensor Networks. *EMSOFT '06: Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*, Oct. 2006.

[10] K. C. Barr and K. Asanović. Energy-Aware Lossless Data Compression. *ACM Transactions on Computer Systems*, 24(3), Aug. 2006.

[11] D. Braginsky and D. Estrin. Rumor Routing Algorithm For Sensor Networks. *WSNA '02: Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications*, Sept. 2002.

[12] N. Brouwers. Darjeeling source code. `https://sourceforge.net/projects/darjeeling`. rev. 398.

[13] N. Brouwers, K. Langendoen, and P. Corke. Darjeeling, A Feature-rich VM for the Resource Poor. *SenSys '09: Proceedings of the Seventh International Conference on Embedded Networked Sensor Systems*, Nov. 2009.

[14] A. Caracas, T. Kramp, M. Baentsch, M. Oestreicher, T. Eirich, and I. Romanov. Mote Runner: A Multi-language Virtual Machine for Small Embedded Devices. *SENSORCOMM '09: Proceedings of the Third International Conference on Sensor Technologies and Applications*, July 2009.

[15] M. Chang and P. Bonnet. Meeting ecologists' requirements with adaptive data acquisition. *SenSys '10: Proceedings of the Eighth International Conference on Embedded Networked Sensor Systems*, Nov. 2010.

[16] Chipcon. CC2420 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver. 2004.

[17] 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.

[18] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent Types for Low-Level Programming. *ESOP'07: Proceedings of the Sixteenth European Symposium on Programming*, Mar. 2007.

[19] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient memory safety for TinyOS. *SenSys '07: Proceedings of the Fifth International Conference on Embedded Networked Sensor Systems*, Nov. 2007.

[20] A. Courbot, G. Grimaud, and J.-J. Vandewalle. Efficient Off-board Deployment and Customization of Virtual Machine-based Embedded Systems. *ACM Transactions on Embedded Computing Systems*, 9(3), Feb. 2010.

[21] Crossbow Technology. MICAz Wireless Measurement System datasheet.

[22] S. Dawson-Haggerty, X. Jiang, G. Tolle, J. Ortiz, and D. Culler. sMAP: a simple measurement and actuation profile for physical information. *SenSys '10: Proceedings of the Eighth International Conference on Embedded Networked Sensor Systems*, Nov. 2010.

[23] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. *OOPSLA '96: Proceedings of the Eleventh ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 1996.

[24] J. Ellul. *Run-time Compilation Techniques for Wireless Sensor Networks*. PhD thesis, University of Southampton, 2012.

[25] J. Ellul and K. Martinez. Run-Time Compilation of Bytecode in Sensor Networks. *SENSORCOMM '10: Proceedings of the Fourth International Conference on Sensor Technologies and Applications*, July 2010.

[26] M. A. Ertl. Stack Caching for Interpreters. *PLDI '95: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1995.

[27] L. Evers. *Concise and Flexible Programming of Wireless Sensor Networks*. PhD thesis, University of Twente, 2010.

[28] C.-L. Fok, G.-C. Roman, and C. Lu. Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications. *ICDCS '05: Proceedings of the Twenty-Fifth IEEE International Conference on Distributed Computing Systems*, June 2005.

[29] A. Gal, C. W. Probst, and M. Franz. HotpathVM: An Effective JIT Compiler for Resource-constrained Devices. *VEE '06: Proceedings of the Second International Conference on Virtual Execution Environments*, June 2006.

[30] 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.

[31] D. George. Micropython. `http://www.micropython.org`, 2017.

[32] 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`.

[33] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java® Language Specification, Java SE 8 Edition*. Feb. 2015.

[34] 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.

[35] L. Gu and J. A. Stankovic. t-kernel: providing reliable OS support to wireless sensor networks. *SenSys '06: Proceedings of the Fourth International Conference on Embedded Networked Sensor Systems*, Nov. 2006.

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

[37] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A Dynamic Operating System for Sensor Nodes. *MobiSys '05: Proceedings of the Third International Conference on Mobile Systems, Applications, and Services*, June 2005.

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

[39] T. Hardin, J. Hester, P. Proctor, J. Sorber, and D. Kotz. Poster: Memory Protection in Ultra-Low-Power Multi-Application Wearables . *MobiSys '17: Proceedings of the Fifteenth International Conference on Mobile Systems, Applications, and Services*.

[40] 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.

[41] K. Hong, J. Park, T. Kim, S. Kim, H. Kim, Y. Ko, J. Park, B. Burgstaller, and B. Scholz. TinyVM: An Energy-Efficient Execution Infrastructure for Sensor Networks. Technical report, Department of Computer Science, Yonsei University, 2009.

[42] C.-H. A. Hsieh, J. C. Gyllenhaal, and W.-M. W. Hwu. Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results. *MICRO-29: Proceedings of the Twenty-Ninth Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 1996.

[43] D. Hughes, K. Thoelen, J. Maerien, N. Matthys, W. Horre, J. Del Cid, C. Huygens, S. Michiels, and W. Joosen. LooCI: The Loosely-coupled Component Infrastructure. *NCA '12: Proceedings of the Eleventh IEEE International Symposium on Network Computing and Applications*, Aug. 2012.

[44] C. Intanogonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. *MobiCom '00 Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking*, Aug. 2000.

[45] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani. A Study of Devirtualization Techniques for a JavaTM Just-In-Time Compiler. *OOPSLA '00: Proceedings of the Fifteenth ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2000.

[46] Y. Kishino, Y. Yanagisawa, T. Terada, M. Tsukamoto, and T. Suyama. CILIX: a Small CIL Virtual Machine for Wireless Sensor Devices. *Pervasive '10: Proceedings of the Eighth International Conference on Pervasive Computing*, May 2010.

[47] J. Koshy and R. Pandey. VM*: Synthesizing Scalable Runtime Environments for Sensor Networks. *SenSys '05: Proceedings of the Third International Conference on Embedded Networked Sensor Systems*, Nov. 2005.

[48] A. Krall. Efficient JavaVM Just-in-Time Compilation. *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, Oct. 1998.

[49] R. Kumar, E. Kohler, and M. Srivastava. Harbor: software-based memory protection for sensor nodes. *IPSN '07: Proceedings of the Sixth International Symposium on Information Processing in Sensor Networks*, Apr. 2007.

[50] K. Langendoen, A. Baggio, and O. Visser. Murphy Loves Potatoes: Experiences from a Pilot Sensor Network Deployment in Precision Agriculture. *IPDPS*

'06: Twentieth International Parallel and Distributed Processing Symposium, Apr. 2006.

[51] B. Latré, P. De Mil, I. Moerman, B. Dhoedt, P. Demeester, and N. Van Dierdonck. Throughput and Delay Analysis of Unslotted IEEE 802.15.4. *Academy Publisher Journal of Networks*, 1(1), May 2006.

[52] P. Levis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. *ASPLOS X: Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[53] P. Levis, D. Gay, and D. Culler. Active Sensor Networks. *NSDI '05: Proceedings of the Second conference on Symposium on Networked Systems Design & Implementation*, May 2005.

[54] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An Operating System for Sensor Networks. *Springer Verlag Ambient Intelligence*, 2005.

[55] K.-J. Lin, N. Reijers, Y.-C. Wang, C.-S. Shih, and J. Y. Hsu. Building Smart M2M Applications Using the WuKong Profile Framework. *iThings '13: Proceedings of the IEEE International Conference on Internet of Things*, Aug. 2013.

[56] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java® Virtual Machine Specification Java SE 9 Edition*. Aug. 2017.

[57] H. Liu, T. Roeder, K. Walsh, R. Barr, and E. G. Sirer. Design and Implementation of a Single System Image Operating System for Ad Hoc Networks. *MobiSys '05: Proceedings of the Third International Conference on Mobile Systems, Applications, and Services*, June 2005.

[58] K. Lorincz, B.-r. Chen, G. W. Challen, A. R. Chowdhury, S. Patel, P. Bonato, and M. Welsh. Mercury: A Wearable Sensor Network Platform for High-Fidelity Motion Analysis. *SenSys '09: Proceedings of the Seventh International Conference on Embedded Networked Sensor Systems*, Nov. 2009.

[59] K. Lorincz and M. Welsh. MoteTrack: A Robust, Decentralized Approach to RF-Based Location Tracking. `http://www.eecs.harvard.edu/~konrad/projects/motetrack`, 2006.

[60] K. Lorincz and M. Welsh. MoteTrack: a robust, decentralized approach to RF-based location tracking. *Springer Personal and Ubiquitous Computing*, 11(6), Aug. 2007.

[61] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Transactions on Database Systems*, 30(1), Mar. 2005.

[62] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless Sensor Networks for Habitat Monitoring. *WSNA '02: Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications*, Sept. 2002.

[63] F. Marcelloni and M. Vecchio. An Efficient Lossless Compression Algorithm for Tiny Nodes of Monitoring Wireless Sensor Networks. *The Computer Journal*, 52(8), Nov. 2009.

[64] Microchip Technology. AVR Instruction Set Manual. 1997.

[65] Microchip Technology. 8-bit Atmel Microcontroller with 128KBytes In-System Programmable Flash ATmega128 ATmega128L. 2011.

[66] G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa: A Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code. *COOTS '97: Proceedings of the Third USENIX Conference on Object-Oriented Technologies*, June 1997.

[67] R. Müller, G. Alonso, and D. Kossmann. A Virtual Machine for Sensor Networks. *EuroSys '07: Proceedings of the Second ACM SIGOPS/EuroSys European Conference on Computer Systems*, Mar. 2007.

[68] D. Niculescu and B. Nath. Ad hoc positioning system (APS). *GLOBECOM '01: IEEE Global Telecommunications Conference*, Nov. 2001.

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

[70] Oracle. *Java Card 3 Platform Virtual Machine Specification, Classic Edition Version 3.0.5*. May 2015.

[71] PhysioNet. PTB Diagnostic ECG Database (ptbdb). `https://www.physionet.org/cgi-bin/atm/ATM`. First 256 samples from the following dataset. Database: PTB Diagnostic ECG Database (ptbdb), Record: patient001/s0010_re, Signals: i, Length: 10s, Time format: samples, Data format: raw ADC units.

[72] K. Pister. On the Limits and Applications of MEMS Sensor Networks. UC Berkeley, 2001.

[73] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling Ultra-Low Power Wireless Research. *IPSN '05: Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks*, Apr. 2005.

[74] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson. Toba: Java For Applications A Way Ahead of Time (WAT) Compiler. *COOTS '97: Proceedings of the Third USENIX Conference on Object-Oriented Technologies*, June 1997.

[75] A. S. A. Quadri and B. Othman Sidek. An Introduction to Over-the-Air Programming in Wireless Sensor Networks. *International Journal of Computer Science & Network Solutions*, 2(2), Feb. 2014.

[76] N. Reijers and K. Langendoen. Efficient Code Distribution in Wireless Sensor Networks. *WSNA '03: Proceedings of the Second ACM International Workshop on Wireless Sensor Networks and Applications*, Sept. 2003.

[77] N. Reijers, K. J. Lin, Y. C. Wang, C. S. Shih, and J. Y. Hsu. Design of an Intelligent Middleware for Flexible Sensor Configuration in M2M Systems. *SENSORNETS '13: Second International Conference on Sensor Networks*, Feb. 2013.

[78] C. Savarese, J. Rabaey, and K. Langendoen. Robust Positioning Algorithms for Distributed Ad-Hoc Wireless Sensor Networks. *USENIX Annual Technical Conference*, Apr. 2002.

[79] A. Savvides, H. Park, and M. B. Srivastava. The Bits and Flops of the N-hop Multilateration Primitive For Node Localization Problems. *WSNA '02: Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications*, Sept. 2002.

[80] N. Shaylor, D. N. Simon, and W. R. Bush. A Java Virtual Machine Architecture for Very Small Devices. *LCTES '03: Proceedings of the ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, June 2003.

[81] N. A. Simakov, M. D. Innus, M. D. Jones, J. P. White, S. M. Gallo, R. L. DeLeon, and T. R. Furlani. Effect of Meltdown and Spectre Patches on the Performance of HPC Applications. Jan. 2018. arXiv:1801.04329.

[82] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java™ on the bare metal of wireless sensor devices: the squawk Java virtual machine. *VEE '06: Proceedings of the Second International Conference on Virtual Execution Environments*, June 2006.

[83] P. H. Su, J. Y.-J. Hsu, K.-J. Lin, Y.-C. Wang, and C.-S. Shih. Decentralized Fault Tolerance Mechanism for Intelligent IoT/M2M Middleware. *IEEE World Forum for Internet of Things*, Mar. 2014.

[84] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1), Jan. 2000.

[85] Team libtom. LibTomCrypt. `http://www.libtom.net/LibTomCrypt`, 2017.

[86] Texas Instruments. MSP430x1xx Family User's Guide (slau049f). 2006.

[87] Texas Instruments. MSP430F15x, MSP430F16x, MSP430F161x MIXED SIGNAL MICROCONTROLLER (SLAS368G). 2011.

[88] The Embedded Microprocessor Benchmark Consortium. CoreMark 1.0. `http://www.eembc.org/coremark`, 2009.

[89] B. L. Titzer. Virgil: Objects on the Head of a Pin. *OOPSLA '06: Proceedings of the Twenty-First ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2006.

[90] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: Scalable Sensor Network Simulation with Precise Timing. *IPSN '05: Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks*, Apr. 2005.

[91] D. M. Tung, N. V. Toan, and J.-G. Lee. Exploring the Current Consumption of an Intel Edison Module for IoT Applications. *I2MTC '17: Proceedings of the IEEE International Instrumentation and Measurement Technology Conference*, May 2017.

[92] P. Tyma. Why are we using Java again? *Communications of the ACM*, 41(6), June 1998.

[93] UCLA Networked & Embedded Systems Laboratory. SOS operating system. `https://github.com/nesl/sos-2x`, 2009.

[94] T. van Dam and K. Langendoen. An Adaptive Energy-Efficient MAC Protocol for Wireless Sensor Networks. *SenSys '03: Proceedings of the First International Conference on Embedded Networked Sensor Systems*, Nov. 2003.

[95] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *SOSP '93: Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, Dec. 1993.

[96] A. Wang, Y.-T. Huang, C.-T. Lee, H.-P. Hsu, and P. H. Chou. EcoBT: Miniature, Versatile Mote Platform Based on Bluetooth Low Energy Technology. *iThings '14: Proceedings of the IEEE International Conference on Internet of Things*, Sept. 2014.

[97] B. Warneke, M. Last, B. Liebowitz, and K. S. J. Pister. Smart Dust: Communicating with a Cubic-Millimeter Computer. *IEEE Computer*, 34(1), Jan. 2001.

[98] N. Weerasinghe and G. Coulson. Lightweight module isolation for sensor nodes. *MobiVirt '08: Proceedings of the First Workshop on Virtualization in Mobile Computing*, June 2008.

[99] M. Weiser. The Computer for the 21st Century. *Scientific American*, Sept. 1991.

[100] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and Yield in a Volcano Monitoring Sensor Network. *OSDI '06: Proceedings of the Seventh Symposium on Operating Systems Design and Implementation*, Nov. 2006.

[101] D. Wheeler and R. Needham. Correction to xtea. Technical report, Computer Laboratory, University of Cambridge, Oct. 1998.

[102] I. Wirjawan, J. Koshy, R. Pandey, and Y. Ramin. Balancing Computation and Code Distribution Costs: The Case for Hybrid Execution in Sensor Networks. *Elsevier Ad Hoc Networks*, 6(8), Nov. 2008.

[103] W. Ye, J. Heidemann, and D. Estrin. An Energy-Efficient MAC Protocol for WirelessSensor Networks. *INFOCOM '02: Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, June 2002.

[104] Y. Zhang, M. Yang, B. Zhou, Z. Yang, W. Zhang, and B. Zang. Swift: A Register-based JIT Compiler for Embedded JVMs. *VEE '12: Proceedings of the Eighth International Conference on Virtual Execution Environments*, Mar. 2012.