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

我的 VM
My VM

賴爾思
Niels Reijers

指導教授：施吉□教授
Advisor: Prof. Chi-Sheng Shih

中華民國 107 年 3 月
March, 2018

國立臺灣大學博士學位論文
口試委員會審定書

我的 VM
My VM

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

口試委員:

_____

_____ _____

_____ _____

_____ _____

_____ _____

所　長: _____

Acknowledgements

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

摘要

中文摘要

關鍵字： 關鍵字

Abstract

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

In this paper we propose MyVM, a sensor node JVM based on the Darjeeling VM, and 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.

MyVM uses Ahead-of-Time compilation to native code to improve performance and introduces a range of optimisations to eliminate most of the overhead still present in previous work on sensor node AOT compilers. Safety is guaranteed by a set of run-time and translation-time checks. The simplicity of the JVM'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.

Using a set of 8 benchmarks with varying characteristic, including the well known CoreMark benchmark, we show this results in an average performance roughly 2x slower than unsafe native code. Without safety checks, this drops to 1.7x. Thus, MyVM combines the desirable properties of existing

work on both safety and virtual machines for sensor networks, while delivering performance and code size overhead comparable or better than existing solutions.

Keywords: keyword

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 has slowed as we approach fundamental physical limits.

While Moore's law is most commonly associated with improvements in performance, it has 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 isn't. However, at certain thresholds, it enables revolutions: 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 [84]. Around the turn of the century, we were able able to scale down useful, working devices to the size of a few millimetres [83]. 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 [4], to precision agriculture [42], habitat monitoring [54], and environmental monitoring [85, 14]. 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 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 features found in modern desktop CPUs, they typically do have several sleep modes, allowing them to reduce power consumption by over 99%. 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 also limited, typically 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 [61]. While the first prediction has come true [82], 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 achieving useful functionality in as small a space as possible and the tradeoffs involved, 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. To make these applications work new protocols were needed 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 [88, 80], lightweight operating systems and virtual machines trading functionality for size and complexity [45, 30, **?**, 43, 13], lightweight routing and data aggregation [37, 12], lightweight

data compression and reprogramming techniques trading CPU cycles for transmitted bits [55, 65], lightweight localisation trading accuracy for complexity [58, 67, 68], etc.

What these all have in common is that they revisit classing 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 also 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 of this information to conserve energy and still make sure the house is at a comfortable temperature when the user gets home.

While IoT and WSN overlap and 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, while IoT devices come in a wide range, with vastly different performance characteristics, cost, and power requirements.

On one end of the spectrum are devices like the Intel Edison and Raspberry Pi. These are the results 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 TI 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 don't work for the other.

Another 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, a user's smart devices cooperate to implement new applications, but these may come from many different vendors and may not be specifically designed for the application a user may run. Coming back to the example from Weiser's paper [84], 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 a smart coffee maker and a smart alarm clock to be programmed in such a way to enable this application.

Thus, many IoT applications are inherently heterogeneous, and as Gu points out, 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 [30]. 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 sent to any node, regardless of it's 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 sensor node VMs, some based on Java and .Net, have also been developed. They manage to pack an impressive set of features on such a limited platform, but almost all sacrifice performance.

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 [21]
TinyVM	Yonsei University	ATmega128	14x-72x slower [33]
DVM	UCLA	ATmega128L	108x slower [10] 500x slower [41]
SensorScheme	University of Twente	MSP430	4x-105x slower [24]

1.2.1 Performance

The VMs for which we have found concrete performance data are shown in Table 1.1. The best case, the 'only' 4x slowdown in one of SensorScheme's benchmarks, is a tiny benchmark that just calls 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 be spent in the CPU compared to communication, or sensors.

But 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 two concrete examples below.

Table 1.2: Energy consumption breakdown for the Mercury motion analysis application. (source: [52])

Component	Energy (uJ)
Sampling accel	2805
CPU (activity filter)	946
Radio listen (LPL, 4% duty cycle)	2680
Time sync protocol (FTSP)	125
Sampling gyro	53163
Log raw samples to flash	2590
Read raw samples from flash	3413
Transmit raw samples	19958
Compute features	718
Log features to flash	34
Read features to flash	44
Transmit features	249
512-point FFT	12920

Mercury Few applications report a detailed breakdown of their power consumption, but one that does is Mercury [52], a platform for motion analysis. The data reported in their paper is copied in Table 1.2. The greatest energy consumer is the sampling of a gyroscope, at 53163 uJ. Only 1664 uJ 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 us that transmitting raw data is a major energy consumer. To reduce this, Mercury has the option of first extracting features from the 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 a 10x to 100x slowdown in the feature extraction algorithm this would not work anymore, since 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 typically one of the major power consumers on mobile devices, compressing data before it is sent is an attractive option to conserve energy spent on transmission, but energy must also be spent on the CPU during (de)compression. Barr and Asanovi´c have analysed this tradeoff for five compression algorithms on the Skiff research platform, with hardware similar to the Compaq iPAQ. They found the break-even point to be at about 1000 instructions for each bit saved. Beyond that the energy spent on compression would start to outweigh the energy saved on transmission. They found a number of cases where this was the case, and using some compression algorithms would increase total energy consumption, compared to sending uncompressed data [11].

Most of the traditional algorithms considered by they Barr and Asanovi´c are too complex to run on a sensor node, so specialised compression algorithms have been developed for sensor nodes. One such algorithm is LEC [55], a simple lossless compression algorithm that can be implemented in only a few lines of 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 from the datasheets for the Atmel ATMEGA128 [9] CPU, we determine the energy per cycle. Running at 8 Mhz and 2.7V, the ATMEGA consumes 7.5mA.

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

We can do a similar calculation to determine the energy per bit for the Chipcon CC2420 radio. This radio can also operate at 3V, 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

$$17.4mA * 2.7V/250kbps = 187.9nJ/bit \tag{1.2}$$

As a result, we can spend $187.9/2.53 \approx 74$ cycles per bit to reduce the size of the transmitted data, and still conserve energy.

We implemented the LEC compression algorithm and used it to compress a dataset of 256 16-bit ECG measurements [60], 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 424 uJ on transmitting the data, at the expense of 246 uJ extra energy spent in the CPU.

Table 1.3: LEC compression energy savings

ATMEGA128 energy per cycle	2.53 nJ/cycle
CC2420 energy per transmitted bit	187.9 nJ/cycle
LEC compression cycles spent	97052 cycles
LEC compression bits saved	2256 bits
LEC compression cycles per bit	43 cycles/bit
energy expended	246 uJ
energy saved	424 uJ
ratio saved/expended	1.7x

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 1.7x more than the extra energy expended on the CPU. This means that if the slowdown from running this in a VM is more than 1.9, and thus time and energy spent on the CPU increases by a factor larger than 1.9, this would tip the balance in favour of just sending the raw data.

Where exactly the 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 [62] 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 chooses to transmit 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 only meant to indicate that there are many situations where the slowdown caused by current sensor node VMs will be an issue. For Mercury's feature extraction, the break-even point is around 27x slowdown, while for this case of LEC compression it is at only 1.9x. While the exact numbers depend on the application, it is clear that a slowdown of one to two orders of magnitude will affect battery lifetime in many cases.

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 environments, but are impractical on sensor node platforms that can often 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. 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 JVM compiler that compiles to our sensor node's native code to test the resulting performance, but we would expect it would be similar to compiled C code. 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 thesis will focus on the middle option: translating the bytecode to native

code on the device itself, at load time. We will build on previous work by Joshua Ellul [21] on AOT translation on sensor nodes, which resulted in a reduction in performance overhead to a slowdown of up to 9x, but also resulted in an increase in the size of the stored programmes of up to 4.5x, limiting the size of the programmes that can be loaded on a device.

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 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 style 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. 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 and 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 say anything about the correctness of the application itself: buggy code may still corrupt it's 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 [17] adds runtime checks to detect illegal writes, and can do so efficiently by analysing the source code before it is compiled. However, this doesn't 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 our 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.

1.3 Scope

The main focus of this thesis is on the suitability of VMs as a way to programme sensor node devices. The main research questions we will answer are:

- How close can an AOT compiling sensor node VM come to native C performance?
- What are the tradeoffs?
- Can a VM be an efficient way to provide a safe, sandboxed execution environment on a sensor node?
- Is Java a suitable language for a sensor node VM, and how may it be improved?

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 thesis focusses specifically on small sensor nodes for which no such standards exists. These platforms have the following characteristics:

- Split data and programme memory: RAM for data, and flash memory for code are separated. While some device can execute code from both RAM and flash [76], others cannot [9].
- Very limited memory: Since RAM consumes energy even when the CPU is in sleep mode, it is typically restricted to 10 KB of RAM or less. More flash memory is usually available to hold programmes, but at 32-256KB this is still very limited.

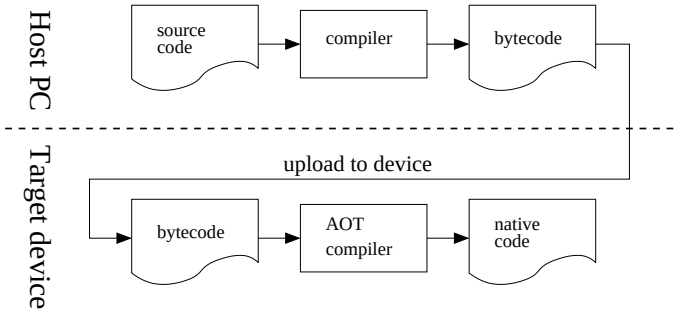


Figure 1.1: Highlevel overview of the compilation process

- Simple CPUs: While usually rich in IO to drive actuators and read from sensors, the rest of the CPUs used in these devices is usually a very simple RISC design to save cost and power consumption. Instructions usually take a fixed number of cycles, since memory is on chip access times are constant and fast, and there are no complicating factors like pipelines, caches, or branch predictions to consider.

- Very limited energy budget: Typical usage scenarios for WSNs 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 compilation process. The host PC will compile some 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 then execute the resulting native code.

We will see that in order to achieve good performance, an optimising compiler to generate high quality bytecode is necessary. However, the focus in this thesis is on developing

techniques for the AOT compiler running on the sensor node. We want to examine what kind of 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 compiler is clearly outside the scope of this thesis, but we will do some manual optimisations that we are confident an optimising compiler could do automatically to determine what level of performance is possible.

Source language Note that in Figure 1.1 we use anonymous '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 Darjeeling VM [13], not because we believe Java to be a particularly good choice.

The question we wish to answer is not "How well can *Java* run on a sensor node?", but "How well can a VM providing *platform independence and safety* run on a sensor node?", and "What should a VM specifically designed for sensor nodes look like?"

We will see that in fact Java, in its current form, is nearly unusable for serious applications, and will end with a number of suggestions on how to remedy these issues.

1.4 Contributions

This thesis makes the following contributions:

- We identify the major sources of overhead when using the baseline approach to AOT compilation, as described by Ellul and Martinez [22, 21].
- Using the results of this analysis, we propose a set of optimisations to address each source of overhead, including a lightweight alternative to Java method invocation to reduce method call overhead.
- These optimisations eliminate most of the performance overhead caused by the JVM's stack-based architecture, and over 80% of performance overhead overall.
- They also reduce the code size overhead by 56%, and show that the increase in VM

size is quickly compensated for, thus mitigating a drawback of the previous AOT approach.

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

- We describe a number of checks that are sufficient for the VM to provide our desired safety guarantees, and show how most of these checks can be easily done at load time, reducing the overhead of run-time checks.

- We provide a comprehensive evaluation to analyse the overhead and the impact of each optimisation, and to show these results hold for a set of benchmarks with very different characteristics, including the commonly used CoreMark benchmark [16], and a number of benchmarks with typical sensor node code.

- Using the same benchmarks we evaluate the cost of our safety checks, and show this to be comparable or better than two existing approaches to provide a safe execution environment on a sensor node.

- Finally, we identify a number of aspects of the Java virtual machine that 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 some necessary background knowledge on wireless sensor networks, Java and the Java virtual machine, and JIT 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 MyVM.

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. In a number of cases where there were multiple options to implement an optimisation we also describe alternatives and motivate our choice.

Chapter 6 presents a set of checks that allow MyVM 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 and compares this with related work on safe execution environments. We use a set of benchmarks with different characteristics, both synthetic ones to show the behaviour in extreme cases and several benchmarks taken from real sensor node code.

Chapter 8 describes a number of issues we encountered while doing this work, which show Java in its current form 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

This dissertation is based on previously published technical reports or conference proceedings. The material presented in the subsequent chapters is based on the content of these publications.

- 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), Feb. 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), Feb. 2017.

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

1.7 Naming

In literature various names are used for our target devices. The word *mote* was common in early WSN research, as was *sensor node*, or *device* although the latter may also refer to larger, more powerful devices. In this thesis we will use the terms *sensor node* or simply *node* interchangeably to refer solely to the type of severely resource-constrained devices described above.

The terms *wireless sensor networks*, *cyber-physical systems* and *internet-of-things* are all commonly used to describe sensor node applications. We will use WSN since it is the longer established term, and more clearly focussed on resource-constrained devices, but note that this also covers a large portion of IoT applications.

When reprogramming these devices, the code is expected to be sent by a more powerful device in charge of configuring the network. These more powerful devices outside or on the edge of wireless sensor network are refered to in literature by various names, including *server*, *gateway*, *master*, *controller*, or *host*. While these names have slightly different accents or refer to different roles, this is not relevant for the work in this thesis. We will use *host* to refer to the source of the code that is uploaded to the sensor node, possibly over multiple hops in the network, and assume the host to be a more powerful device with desktop-class processing capabilities.

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 also 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 a VM may offer 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 Powerful 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, with the Raspberry Pi Zero being only 65x30mm, and the Intel Edison at 35.5x25mm.

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, the can perform much more complex tasks, but the smallest devices in this class are still significantly larger than the smallest sensor nodes, more expensive, and most importantly, consume significantly more power.

Since devices in this class are capable of running normal, well established VMs, we do not consider them in this thesis, 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 are capable of running for weeks or months on a single battery charge. As a typical example, the MICAz node [74] uses only 30mA when active and 16uA in sleep mode. More recently, the Arduino family of devices, based on similar hardware, has led to a very active community 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,

Table 2.1: Main characteristics of the ATMEGA128 and MSP430F1611 CPUs.

	ATMEGA128 [9, 8]	MSP430F1611 [76, 75]
Number of registers	32	12
Register size	8-bit	16-bit
RAM	4KB	10KB
Flash	128KB	48KB
Frequency	up to 16MHz	up to 8MHz
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]	7.5mA	4.0mA
Deep sleep power consumption	0.3uA	0.2uA
Execute code from	Flash	Flash and RAM

[a] at 8MHz and 2.7V

and physical packages, but the underlying architecture is similar for all. Table 2.1 lists the main characteristics for two popular member 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 thesis.

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. Flash memory is typically in the range of 16KB to 256KB, while RAM, which consumes energy even in sleep mode, is restricted to up to 10KB. There are no caches, and since both memories are on chip, access times are constant and take only a few cycles.

Simple RISC architecture These CPUs achieve their extremely low cost and power consumption by restricting themselves to very simple design. Each instruction costs 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 [45] which provides several basic services for IO, communication and task management. Contrary to a normal OS, TinyOS doesn't 'load' an application, but is in fact more like a library that 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 applications [64]. In some cases these allow the entire application code to be replaced, including the reprogramming protocol itself [65]. In other cases the reprogramming system may be permanent and include other basic services, which is more close to an operating system. 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 also 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 usually *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 of the details relevant to this work.

The first public release of Java was in 1995. It consists of two separate but 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 could be built for any hardware platform, 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 ran applets 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 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 VMs 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, `a` and `b` 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               a
4 ILOAD_2               a   ,   b
5 IADD                  a+b
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. There 53 are simple arithmetic or bitwise operations, such as `IADD` in the example, also in different variations for different datatypes. There 39 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 some 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 suitable basis for a sensor node VM, where we don't have the resources to process anything much more complicated.

2.2.2 Memory

The JVM stores information in three different places:

- The heap: objects and arrays are stored here, and automatically garbage-collected when no longer used.
- The stack frame: each method's stack frame contains a section for that method's operand stack, and for 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 JVM is a 32-bit machine. All the places where data may be stored, objects on the heap, operand stacks and local variable slots, and a class' static variables, are essentially a block of 32-bit wide slots. 64-bit `long` and `double` types occupy two slots on the operand stack of local variable space, 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 *reference*. 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.

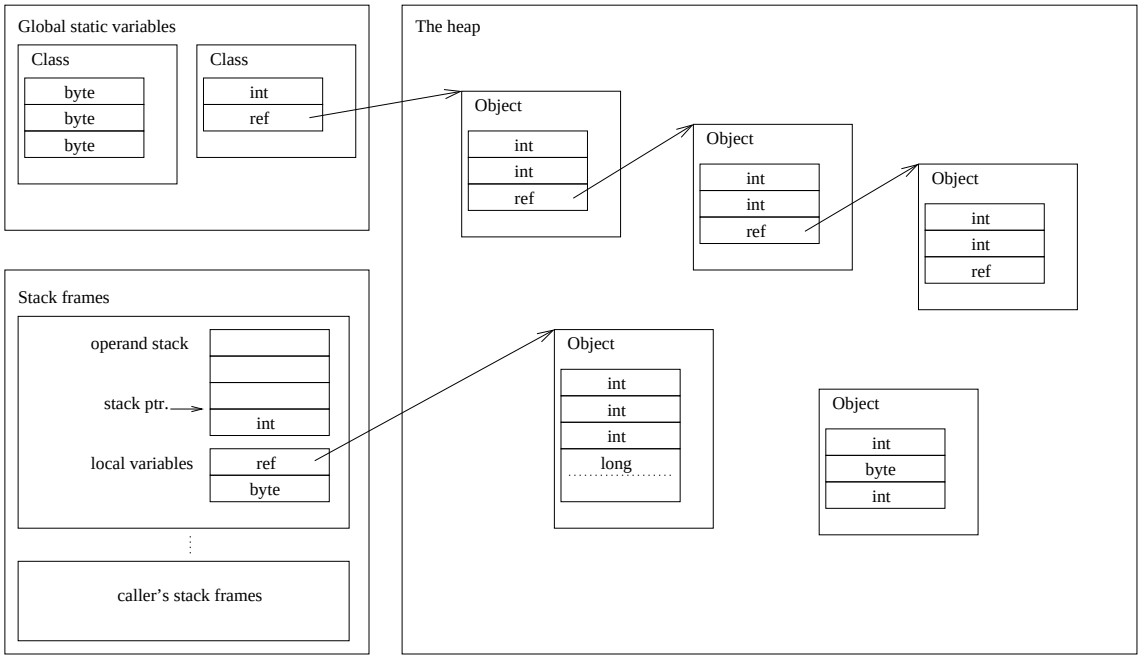


Figure 2.1: Highlevel overview of JVM memory design

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 will be 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 [48]. It performs many checks, for example that branches are within the bounds of the code array for the method, and branch to the beginning of an instruction, that execution cannot fall off the end of the code, 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 stack depth and type of values on a

method's operand stack is known at any point and doesn't over- or underflow, etc.

2.2.4 JIT and AOT compilation

While the popularity of Java rose quickly after its introduction, it also very quickly got a reputation to be 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." [79].

The main reason is all early implementations of the JVM were interpreters. An interpreter executes a program 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 the 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 in at what point the bytecode is translated to native code.

Compilation 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 [63]. Some systems first translate to C [20], 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 during 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 is early versions of Android's ART runtime, which translates an app 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 compiling 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 of compilers, called 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 runtime, a JIT compiler can take advantage of the observed runtime 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 [38].

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 thesis. 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 safey and efficiently to support these applications?

This thesis 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 minimum operating system, such as TinyOS [45]. This results in applications being

programmed from the individual node's perspective, rather than allowing the application to express globally what it 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, were 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 [19], which provides an attractive and flexible RESTful interface 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. TinyDB [53] makes an entire network of sensor nodes available through a SQL-like query language.

ADAE [14], 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's likely a skilled engineer is still needed to translate the user's requirements into the constraint optimisation problem ADAE uses as input.

While in the previous two systems, applications run on a centralised host and simply control the nodes in the network, in Agilla [25] programmes are more distributed and consists of software agents that can move around autonomously in the network. 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 which is based on the

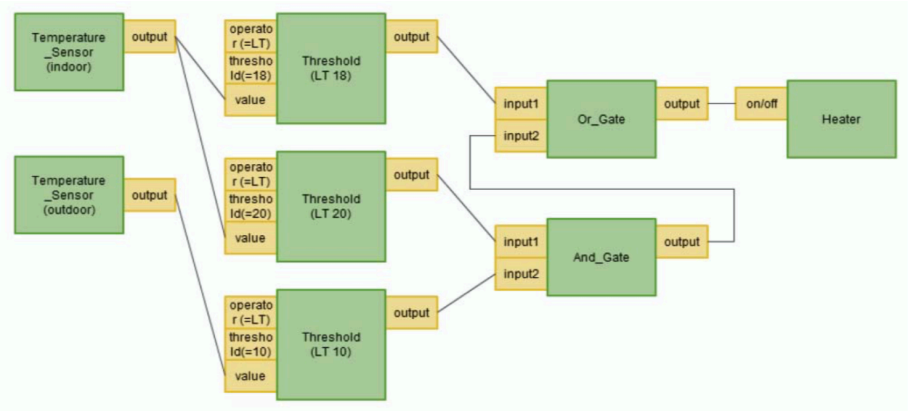


Figure 3.1: Example WuKong flow based programme

Maté VM [43] discussed below makes it hard to use.

LooCI [35] is a component infrastructure middleware for WSNs with standard support for runtime 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. However, it considers component selection to be higher-level services outside of its core functionality.

Cornell's MagnetOS [49], 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. We consider this to be a possible future direction for our study if and when all future WSNs are proved to be more powerful.

3.1.1 WuKong

A final example that we will look at a bit more detail at is WuKong [66, 47].

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 has 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 on.

For a node's hardware components, the node creates WuObjects to represent them 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.

Software components, like the Threshold or And_Gate in the example, 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].

An WuClass is defined by and two C functions or Java methods:

- 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 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 it's tasks may change dynamically while the application is running if the master decides to reconfigure the network, for example in response to failure [73] or because network conditions change.

3.2 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 [44] that provide specialised instructions for a specific problem domain. As an example, designed specifically for WSN, Maté [43] was one of the first to prove VMs can be made to run on sensor nodes. It provides single instructions for tasks that are common on a sensor node, so programmes can be very short.

SwissQM [56] 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* [39] 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 or 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 support are typically not supported.

The Python-on-a-chip project [2] developed a Python bytecode interpreter small enough to fit on sensor nodes. Requiring 55K of programme memory and a recommended 8K of RAM, it fits on many, but not all of the current sensor boards. MicroPython [28] requires slightly more hardware at 256KB programme and 16KB RAM, but has it's own hardware platform and runs Python 3.

The smallest official Java standard is the Connected Device Limited Configuration [59], but since it targets devices with at least a 16 or 32-bit CPU and 160-512KB of flash memory available, it is still too large for most sensor nodes. The available Java VMs for sensor nodes all offer some subset of the standard Java functionality, occupying different points in the tradeoff between the features they provide, and the resources they require.

Darjeeling [13] from Delft University of Technology is able to run modified Java byte-code on sensor nodes, with only minor restrictions. It has no support for floating point, 64-bit datatypes, reflection or synchronised method calls, although the later is not really a restriction since synchronised blocks are supported. Similarly TakaTuka [7] is a slightly more complete and more complex JVM, also running on both AVR and MSP based sensor nodes and offering it's own debugger. Another unique property is it'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 [69] is significantly larger than Darjeeling and TakaTuka, but offers full CLDC compliance.

The opposite approach is taken by NanoVM [32], 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.

Finally, SensorScheme [24] implements a fully functioning LISP interpreter in under 41KB.

3.2.1 Darjeeling

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

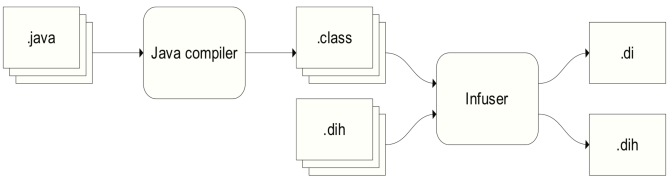


Figure 3.2: Darjeeling infusion process, taken from [13]

Split VM architecture Like most other sensor node JVMs, it uses a *split VM architecture* [71]. The actual virtual machine running on the node does not use standard JVM class files, but these class files first have to be 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 flattening the Java classes into a list of statically 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 it to find the numbered identifiers of classes and methods in the referenced infusion. This is shown in Figure 3.2.

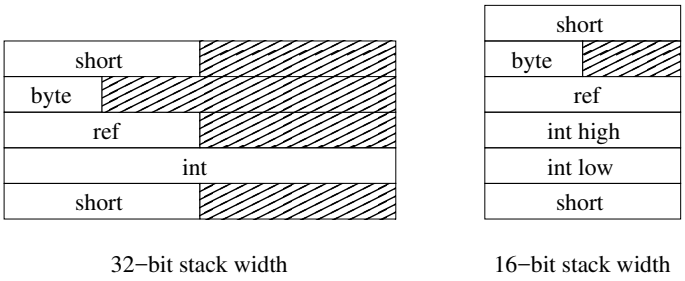


Figure 3.3: Unused memory for 32 and 16-bit stack width

16-bit architecture Besides statically linking classes, the infuser also makes several changes to the bytecode format. Since references on sensor nodes are usually 16-bit, and the use of 8-bit and 16-bit integer variable 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 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. This means 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.

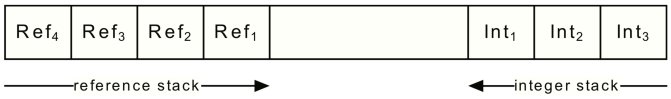


Figure 3.4: Darjeeling split operand stack, taken from [13]

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 it's 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 pointer, 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 byte code 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 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. And it does not support the `synchronized` modifier on static methods.

3.3 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 [21] reports measurements on the TakaTuka VM [7] where the VM is 230x slower than native code, and consumes 150x as much energy. TinyVM [33] is between 14x and 72x slower than C, for a set of 9 benchmarks. DVM [10] has different versions of the same benchmark, where the fully interpreted version is 108x slower than the fully native version. Finally, SensorScheme [24] 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 practice 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 [20, 57]. 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 [18]. 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 key advantages of using a VM.

Hsieh et al. describe an early ahead-of-time compiling desktop Java VM [34], 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 [40]. More recently these efforts have included JIT compilers targeted specifically at embedded devices. Swift [89] 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 [26] has lower requirements, but at 150KB 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 ample space for other tasks 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 [5, 87].

3.3.1 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 tried by Ellul and Martinez [22, 21], and our work builds on their approach.

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

Table 3.1: Example of Ellul's Bytecode to Native Code Translation of `c=a+b;`

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`

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

Peephole optimisation

To reduce this overhead, Ellul proposes a simple peephole optimiser [21] which does 4 optimisation, for each which an examples is shown in Table 3.2. The most important are removing a push immediately followed by a pop to the same register, since this has 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.

Resulting performance

Ellul's approach improves performance considerably compared to the interpreters, but using the standard CoreMark benchmark [16], it still generates code that is still up to 9x slower than optimised native C. The average slowdown found in our benchmarks was

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, taken from [21]

5.4x.

It is important to further improve this for two 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 [14] 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.

Second, reduced performance means the CPU has to stay active for a longer time, resulting in increased cpu power consumption and reduced battery life. If we look back to the data in Section 1.2.1, we see in Table 1.2 that the 'Computer features' optimisation, which reduces the amount of data to be transmitted by over 98% is still a net gain, even if a 8x slowdown would mean it takes 6462uJ to compute the features. However it makes the CPU a large energy consumer which would affect battery life if the application didn't use such an expensive sensor as a gyro. Looking at the FFT, this is already a major energy consumer, and would become the largest one at only 4.1x slowdown.

Looking at the calculations for lossless compression, also in Section 1.2.1, we saw that in this particular case, the ratio of the energy saved on radio transmission vs energy spent on compression was about 1.9:1. 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

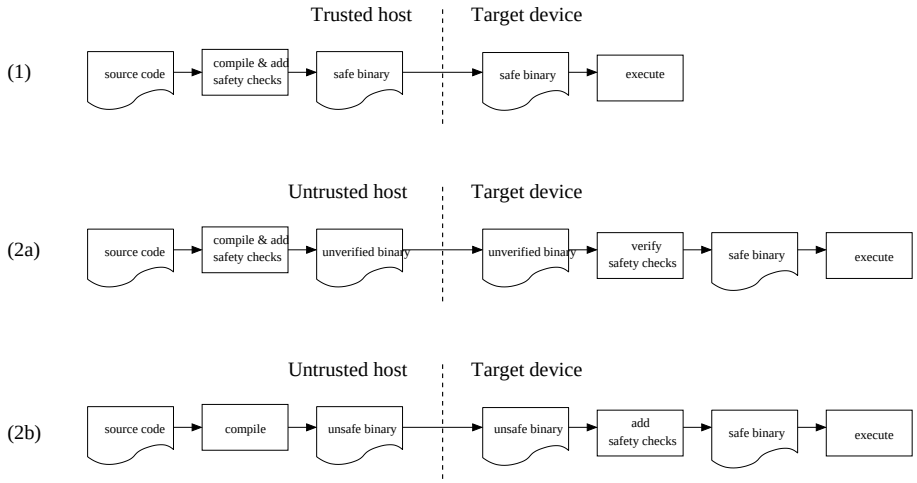


Figure 3.5: Three approaches to provide a safe execution environment

slowdown push some situations past the break-even point, or reduces the benefits of compression in others.

Regardless of these concerns, there are scenarios that may not be able to tolerate the one to two orders of magnitude slowdown seen in interpreters, a slowdown of up to 9x for Ellul's AOT approach may be acceptable. However, there is a second reason to want to improve on it, which is it also results in significantly larger code: on average 2.8x larger. 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 on sensor nodes.

3.4 Safety

With some exceptions [24], most current sensor node VMs don't 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 various levels of 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 runtime safety checks where needed, before sending it to the node. If the node could receive untrusted code from untrusted sources, for example to support mobile agents as in Agilla [25], it needs to guarantee safety independent of whatever code

it may receive from the host to guard against malicious attack.

The first can guard against programming errors, but not against malicious attacks unless the node has some, potentially expensive, authentication mechanism to make sure it only accepts code from trusted sources. The latter is obviously the stronger guarantee, but it also comes at a higher price.

3.4.1 Source code approaches

In the first category are systems that guarantee safety at the source code level. Virgil [77] 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 on the other hand, works on annotated nesC TinyOS code. It uses the Deputy [15] 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

Both approaches eventually result in standard C, which is then compiled and sent to the node. Therefore, both approaches may protect against accidental programming errors, but not against malicious code an attacker may send to it.

3.4.2 Native code approaches

For desktop applications, Wahbe et al. described software fault isolation [81] 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 can't bring down the whole application, but running it as a separate process would incur high overhead

due to frequent context switches. Two basic methods are described to isolate such code from the main application: we can either rewrite the native code at load time, inserting checks at all potentially unsafe writes, or we can compile the code to a more restricted format with the appropriate checks already in place, and verify the code adheres to this standard at load time.

Since we don't 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 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 [41], 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 [31] is then extended with a binary verifier to verify incoming binaries. 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, where the authors note the benchmark producing the highest slowdown is more typical of sensor node code. 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.

Chapter 4

MyVM

This chapter we introduce MyVM. Our VM is based on Darjeeling [13]. Like other sensor node VMs, Darjeeling is originally an interpreter. We replace this interpreter with an AOT compiler: instead of interpreting the bytecode, the VM translates it to native code at load time, before the application is started. While JIT compilation is possible on some devices [21], it depends on the ability to execute code from RAM, which many embedded CPUs, including the ATMEGA, cannot do.

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 will produce code that both performs well, and adds as little code size overhead as possible. In addition, we want our VM to fit as many scenarios as possible, for example scenarios were 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 3.1.1 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 object to another node. This means new Java code may have to be loaded onto a device, while another (part of the) application is already running.

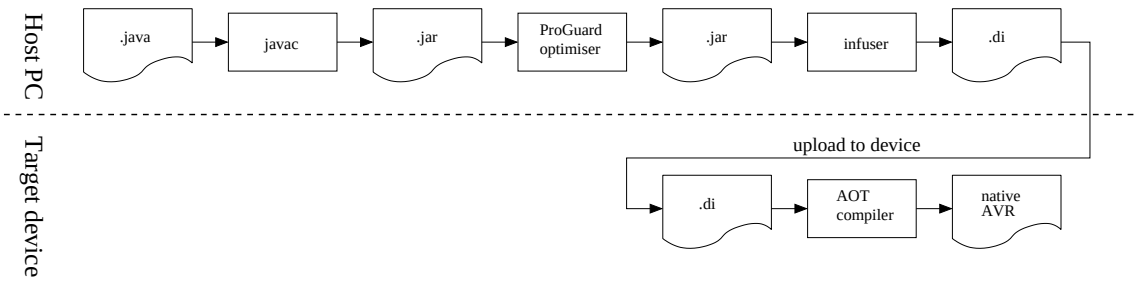


Figure 4.1: Java to native AVR compilation

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 very scarce resource and any memory used by the AOT compiler cannot be used for a different task running at the same time. 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 do more complex optimisations to achieve even better performance through more complex optimisations, but as we will see, quite a lot can be achieved with such 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 also larger than JVM bytecode. This is the price we pay for increased performance, but the optimisations we propose do significantly reduce this code size overhead compared to previous work, thus reducing an important drawback of previous AOT compilation techniques.

4.1.1 Compilation process

The complete process from Java source to a native application on the node is shown in Figure 4.1. Like all sensor node JVMs, Darjeeling uses a modified JVM bytecode. Java source code is first compiled to normal Java classes. We then use ProGuard [72] to do some simple optimisations, but these are very basic steps like removing dead code, and

overlapping local variable slots where possible, etc.

The optimised Java classes are then transformed into Darjeeling's own bytecode format, called an 'infusion'. For details of this transformation we refer to the Darjeeling paper [13]. 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 out 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. Our approach allows the bytecode to be translated to native code in a single pass, on 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.

This means we can free each message, which can be as small as a single JVM instruction, 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 our translation process does not require it to do so, and a protocol that values memory usage over retransmissions cost could simply discard out of order messages and request retransmissions when necessary.

4.2 Translating bytecode to native code

Our baseline AOT compiler is an implementation of Ellul's approach, discussed in Section 3.3.1. We adapt this for the Atmel ATMEGA128 CPU, while Ellul's work uses the Texas Instruments MSP430.

In this unoptimised version of our 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 JVM stack. An example of this is shown in Table 1.

The first column shows a fragment of JVM code which does a shift right of variable `A`,

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

JVM	AOT compiler	AVR	cycles
0: BRTARGET(0)	*« record current addr »*		
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 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)

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 JVM instruction. Together, the first and second column match the case labels and body of a big switch statement in our 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 loading the high and low bytes of a short. The cycle count is based on the actual number of generated instructions, and for a single iteration.

4.2.1 Branches

Forward branches pose a problem for our 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 it records the address where the next instruction will be emitted in a separate part of flash. 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.

4.2.2 Darjeeling split-stack architecture

When Darjeeling's garbage collection runs, it needs to mark the *root set*: the set of all live, directly reachable objects. This is then iteratively expanded to also cover all indirectly

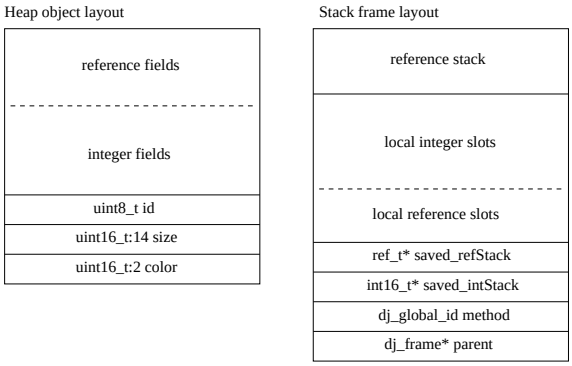


Figure 4.2: Object and stack frame layout

reachable objects. For example in Figure 2.1, only the two objects on the left 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 garbage collected.

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, Darjeeling separates references and integers 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 our AOT compiler we use the native stack for the JVM integer operand stack, so the integer operand stack is not in a method's stack frame, but space for the reference stack still 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.

4.2.3 Peephole optimisation

Since our baseline should be as close as possible to Ellul's MSP430 implementation, we implement the same peephole optimisations. However, differences between the ATMEGA and MSP430 instruction sets means they are not completely identical. The complete set of peephole optimisations is shown in Table 4.2.

Table 4.2: MyVM's peephole optimisations

Before Instructions	Cycles	Length (bytes)	After Instructions	Cycles	Length (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.

A `push` directly followed by a `pop` are 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 optimise blocks of pushes followed by 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.

4.2.4 Bytecode modification

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

- The new `BRTARGET` opcode marks targets of branch instructions, and all branch instructions are modified to target a `BRTARGET` id instead of a bytecode offset.
- The new `MARKLOOP` opcode marks inner loops and the variables they use.

- `_FIXED` versions of the GETFIELD_A and `PUTFIELD_A` opcodes are used to access an object's reference fields when the offset is known at compile time (the offset it always known at compile time for integer fields).
- The new `SIMUL` opcode does 16x16-bit to 32-bit multiplication.
- Array access opcodes now use 16-bit indexes.
- New `_CONST` versions of all (variable) bit shift opcodes to support shift by a constant number of bits.
- The new `INVOKELIGHT` opcode for an optimised 'lightweight' way of calling methods.

4.3 Limitations

Since our compiler 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 before on a sensor node AOT compiler [21], 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.

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.

4.4 Target platforms

We implemented our VM for the ATmega128 CPU. The AVR family of CPUs is widely used in low power embedded systems. 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, with 12 32-bit general purpose registers, or the MSP430, with 12 16-bit registers, and used by Ellul and Martinez [22], to both be good matches as well. In Chapter 7 we will examine the expected impact of the number of registers and of using a 16 or 32-bit architecture.

Chapter 5

Optimisations

Having introduced our baseline AOT compiler, this chapter we will propose several optimisations to improve it's 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 of our baseline approach is still far behind that of optimised native C. To improve performance, 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 code generation overhead
  - Push/pop overhead
  - Load/store overhead
  - JVM instruction set limitations
- Method call overhead

We will briefly discuss each source of overhead 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 typically 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` could. However, since our AOT compiler simply replaces each instruction with a native equivalent, this leads to significant overhead.

We do use the ProGuard optimiser [72], 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 even higher, since more values will be on the stack at the same time. This means more corre-

sponding push and pop instructions will not be consecutive, and the 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 often the same value is needed again soon after. In this case, because the value is no longer on the stack, we need to do another load, which will result in another read from memory.

In Table 4.1, it is clear that the `SLOAD_0` 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 JVM 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 on a sensor node we do not have such resources and must simply execute the instructions as they are.

In Table 4.1 we see that there is no way to express a single bit shift directly. Instead we have to load the constant 1 onto the stack and execute 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 array access. In JVM bytecode each array access will consume the array reference and index from the stack. When looping over an array, this means we that for each iteration we have to load the reference and index back onto the stack again, and redo the address calculation. In contrast, the native C version would typically just slide a pointer over the array.

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, program counter, reference to previous activation frame, and the like)" [29]

Darjeeling'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 Darjeeling 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 int and ref 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
   (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 550 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.

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	`SIMUL` instruction
		`GET/PUTFIELD_A_FIXED` instructions
		constant bit shift optimisation
		16-bit array indexes
Section 5.4	Method call overhead	`INVOKELIGHT` instruction

5.2 Manually optimising the Java source code

As shown in Section 4.1.1, our current implementation uses three steps to translate Java source code to Darjeeling bytecode: the standard Java compiler, the ProGuard optimiser, and Darjeeling's infuser. None of these do any complex optimisations.

In a future version, ProGuard and the infuser should be merged into an 'optimising infuser' which uses all the normal, well-known optimisation techniques to produce bet-

ter 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, we manually optimise the Java source 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 many of these to happen automatically, and without this step it would be impossible to reproduce our results.

We have been careful to limit ourselves to 'fair' optimisations, by which we mean optimisations that an optimising infuser could reasonably be expected to do automatically, given some basic, conservative assumptions about the performance model.

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 inlining small methods.
- Prefer to use 16-bit variables for array indexes where possible.
- Use bit shifts for multiplications by a power of two.

We will briefly examine the effect of some 'unfair' optimisations on the CoreMark benchmark in Section 7.2. These are optimisations that the infuser most likely couldn't do automatically, but a developer who's aware of the performance of the VM could easily do manually.

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 of-

ten only needs to be reused only once for it to be faster to store in a local variable than to calculate it twice. 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 free.

Manual inlining We 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`. ProGuard can also inline small methods, 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 compiler should be able to come to the same result automaticallly.

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 would work well for most devices in this class.

Example An example of these manual optimisations, applied to the bubble sort benchmark, can be seen in Listing 3. 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.

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

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

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

Table 5.2: Improved peephole optimiser

JVM	AOT compiler	AVR	cycles
0: BRTARGET(0)	*« record current addr »*		
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 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)

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:

```
1  PUSH Rs
2  ..
3  ..       instructions in between: - contain the same number of push and pop instructions
4  ..                                - contain no branches
5  ..                                - do not use register Rd
6  ..
7  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

5.3.2 Simple stack caching

Table 5.3: Simple stack caching

JVM	AOT compiler	AVR	cycles	cache state R1	cache state R2	cache state R3	cache state 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	*	*		

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 [23] to eliminate most of the remaining push/pop overhead. Stack caching is not a new technique, originally proposed for Forth interpreters in 1995. But the tradeoffs 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 load 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 1KB 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 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 all registers available for stack caching are in use.

In the baseline AOT approach, each JVM 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:

- `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, `getfree` spills the register that's lowest on the stack to memory by emitting a `PUSH`, and then returns that register. This way the top of the stack is kept in registers, while lower elements may be spilled to memory.

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

- `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 `getfree`, or `pop`.

Using stack caching, code generation is split between the instruction translator, 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 we translate the same example we used before, 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 three 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 cache in the same way as the integer stack. In the next two optimisations we will extend the cache state further.

The example only shows three registers, but the ATmega128 we use has 32 8-bit registers. Since Darjeeling uses a 16-bit stack, we manage them as pairs. 10 registers are reserved, for example as a scratch register or to store a pointer to local or static 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, we flush the whole stack to memory whenever we encounter either a branch or a `BRTARGET` instruction.

This may seem bad for performance, but fortunately in the code generated by `javac` the stack is empty at almost all branches. The 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.

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.

Table 5.4: Popped value caching

JVM	AOT compiler	AVR	cycles	cache state R1	cache state R2	cache state R3	cache state 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		

We add a 'value tag' to each register's cache state to keep track of what value is currently held in the register, even after it is popped from the stack. Some JVM 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 JVM 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 in 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 `pop` has been changed to `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 `pop_tostore` instead of `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 line 5, we need to load LS0 again, but now the cache state shows that LS0 is already in R1. This means we do not need to load it from memory, but 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 we load a value that's already on the stack, we generate a move 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, we do know the state of the registers in the next instruction.

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 that overhead still remains. 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

Table 5.5: Mark loops

JVM	AOT compiler	AVR	cycles	cache state R1	cache state R2	cache state R3	cache state R
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		

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 inner loop. `MARKLOOP` has a larger payload than most JVM 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. We also need to make a small change to `sc_pop_destructive`. If the register we're about to pop is pinned, we cannot just return it since it would corrupt the value of the pinned local variable. Instead we will first emit a move to a free, non-pinned register, and return that instead.

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. Next, `SUSHR_CONST` pops destructively. We cannot simply return register 1 since that would corrupt the value of variable LS0, so `sc_pop_destructive` emits a move to a free register and returns that register instead. Since LS0 is pinned, we can also skip `SSTORE_0`, but we do need to emit a move back to the pinned register.

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

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

5.3.5 Instruction set modifications

Next, we introduce four optimisations that target the type 3 overhead: cases where limitations in the JVM instruction set means we cannot express some operations as efficiently as we would like. 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 optimisa-

tions 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 if we want 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. We extend the infuser 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, we made the following four other modifications to Darjeeling's instruction set:

Constant bit shifts

Table 5.6: Constant bit shift optimisation

JVM	AOT compiler	AVR	cycles	cache state R1	cache state R2	cache state R3	cache state
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	*	
	« it_LSR(operand_1) »	LSR R3	2	LS0PIN	LS1PIN	*	
	« _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		

Almost all 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: we first need to push the constant onto the stack, and then the bit shift is implemented as a simple loop which shifts one bit at a time. If we already know the number of bits to shift by, we can generate much more efficient code.

Note that this is different from other arithmetic operations with a constant operand. For operations such as addition, our translation process results in loading the constant and performing the operation, 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 runtime, a loop must be generated to shift one bit at a time, which is much slower than the code we can generate 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 470 bytes to the VM, but it improves performance, sometimes very significantly, for all but one of our benchmarks.

Surprisingly, when we first implemented this, one benchmark performed better than native C. We found that `avr-gcc` does not optimise constant shifts in all cases. Since our 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

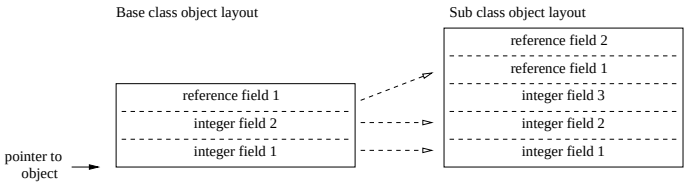


Figure 5.1: Base class and sub class layout

could easily be added. We implemented a version that is close to what `avr-gcc` does, but never better. We only consider cases optimised by `avr-gcc`. For these, we first emit whole byte moves 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, which appears to be up two from the 18 cycles spent using only popped value caching. 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 of Darjeeling's split architecture, 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 subclass instances, so `GETFIELD_A` and `PUTFIELD_A` must examine the object's actual type 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 class 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 compile time. Conveniently, one of the optimisations ProGuard does, is marking 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 Darjeeling's split architecture 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 easy 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 benchmark could be marked `final`. But 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 Darjeeling's split architecture has a lot of 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-bitx16-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 a device with a shorter word size, 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 added more instructions, for example `SIADD` instruction for addition, `BSMUL` for 8-bit to 16-bit multiplication, etc. But there is always 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

Finally, normal JVM array access instructions (`IASTORE`, `IALOAD`, etc) expect the index operand to be a 32-bit integer. 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 expect a 16-bit index instead. This is easily done in Darjeeling's infuser, which contains a specification of the type of operands of each opcode, and will automatically emit type conversions where necessary.

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.

5.4 Method calls

Finally, we will look at the overhead caused by method calls. In native code, the smallest function call only has 8 cycles of overhead for a `CALL` and `RET`, and some `MOV`s may be needed to move the parameters to the right registers. More complicated functions may spend up to 76 cycles saving and restoring call-saved registers. As we have 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 550 cycles, and this increases further for large methods with many parameters.

When we look at the methods in a programme, we typically see a spectrum from a

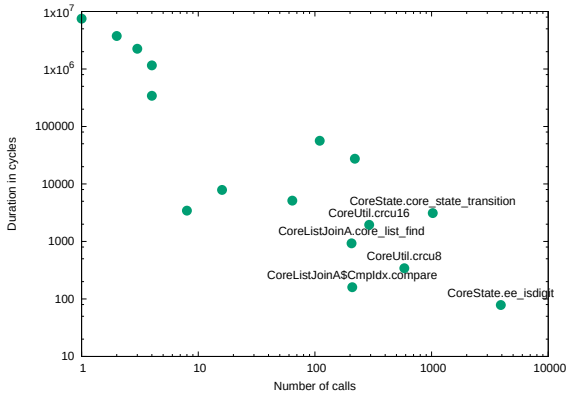


Figure 5.2: CoreMark method calls vs duration with logarithmic scales

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 impact of the method call is not very significant for the overall execution time and we can afford to take the 550 cycles penalty. 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 we have tiny helper functions that may be inlined. However, this is only possible for small methods, or methods called from a single place. In CoreMark's case, `ee_isdigit` was small enough to inline. When we inline larger methods, the tradeoff is an increase in code size. So we have 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:

- We do not create a stack frame for lightweight methods, but use the caller's frame.
- 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, we directly `CALL` the method's code. We bypass the VM completely, reusing the caller's stack frame, and leaving the parameters on the (caller's) stack. In effect, the lightweight method behaves similar to inlined code, but since we can `CALL` it from multiple places, we 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 states, we do have to flush the stack cache 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 4:

```
// 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 4: Simple, stack-only lightweight method example

The normal implementation has a single local variable. It expects the parameter to be stored there 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.

We added a new instruction, `INVOKELIGHT`, to call lightweight methods. In the bottom half of Listing 5 we see 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 5: 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 we want a lightweight method to be able to use local variables, we need to reserve space for them in the caller's stack frame, equal to the maximum number of slots needed by all the lightweight methods it may call.

In our AOT compiled code, we use 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.

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 lightweight methods to call other lightweight methods. A good example from the CoreMark benchmark is the 16-bit `crcu16` function, which is implemented as two calls to `crcu8`. While `crcu8` is the most critical, there is still one call to `crcu16` for every two to `crcu8`.

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 JVM integer stack value, which means the operands to a lightweight method will be on the native stack. But when we do a `CALL`,

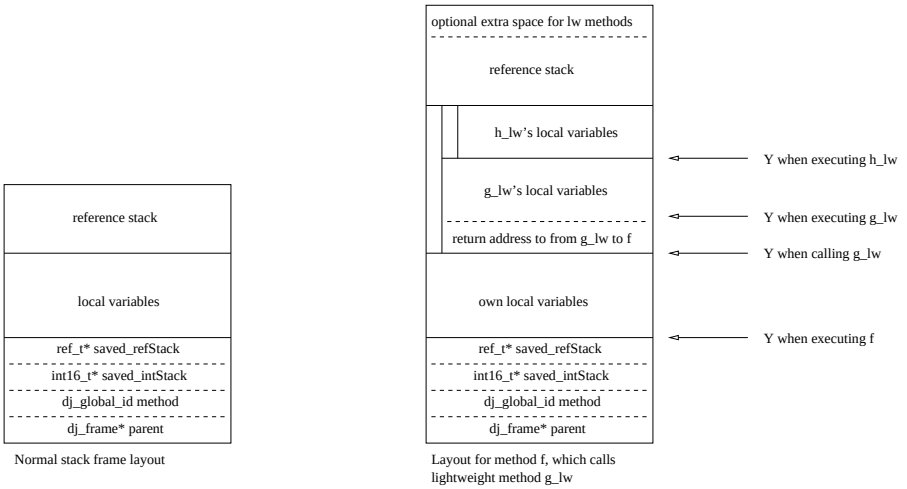


Figure 5.3: Stack frame layout for a normal method `f`, which calls lightweight method `g_lw`, which in turn calls lightweight method `h_lw`.

the return address is put on the stack, covering the method parameters.

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 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 we want.

This difference in method prologue and epilogue is the only difference in the way the VM generates code for a lightweight method, all JVM 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 descriptor.

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 at 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 its 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. The only case where this would be necessary is when it is called inside a MARKLOOP block that uses the same register to pin a variable. In this case we save those variables back to memory before calling the lightweight method and load them 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 mark loop 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 6, 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 we see we first flush the cache to memory, and then save the value of the local short at offset 22 that was pinned to R14:R15. Finally we add 26 to the Y register to skip the caller's own local variables and point Y to

the start of the space reserved for lightweight method locals.

In the method call, we first see the return address is popped off the stack into a register. Since `g_lw` may call another lightweight method, we cannot leave it there but store it in the first local slot, incrementing Y in the process. After `g_lw` is done, we see the reverse process to return to the caller, where we then see 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 register.

```
// 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 locals
 call &g_lw
 sbiw Y, 26       // Restore Y
 ldd r14, Y+22    // Reload pinned value
 ldd r15, Y+23
```

```
// 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
```

Listing 6: Full lightweight method call

5.4.2 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 currently slightly expensive because of the way 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 may mean we may not be able to skip as many `LOAD` instructions after the lightweight call, but 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. We just need to save the return address and restore it in the epilogue, which takes 8 cycles if we can leave it in a register, or 16 if we need to store it 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 7. 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, so the relative overhead is still acceptable.

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 aren't 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.

Comparing this to a normal method call, we see 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 4) are fixed, meaning a call will cost at least around 550 cycles, increasing to over 700 cycles for more complex methods taking many parameters.

5.4.3 Creating lightweight methods

We currently support two ways to create a lightweight method:

- handwritten JVM bytecode
- converting a Java method

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 [a]			8 per word	8 per word	8 per word
`INVOKE`			8 to 68	8 to 68	~82
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*	*~542 to ~603 + 12 or 16 per word*

[a] excluding effect on future popped value cache performance because of cleared value tags

Handwritten JVM 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 JVM bytecode, which the infuser will simply add to the infusion, and then process it in the same way as it processes a normal method, with one step added:

For lightweight methods, the parameters will be on the stack at the start of the method, but the infusers 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.

Converting 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 developed 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 JVM 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 7 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.

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

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

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

Listing 7: 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`, 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.

5.4.4 Limitations and tradeoffs

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

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, we do

not support recursion, although lightweight calls can be nested.

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 we're in a lightweight call, 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.

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 We currently do not support lightweight virtual methods, since the overhead of resolving the target of the invoke is large compared to the rest of the invoke overhead, but this is something that could be considered in future work.

Stack frame usage Finally, while many methods can be made lightweight, we should remember that a method calling a lightweight method will always reserve 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 if we have 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`, we will have reserved space for `big_lw` twice, both in `f1`'s and in `f2`'s frame.

Chapter 6

Safety

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

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

1. write to memory outside the areas assigned to it,
2. execute code it does not have permission for, or
3. retain control of the CPU indefinitely

Given the first two, the last guarantee is easy to implement: the VM can simply set a timer to trap back to the VM after a certain amount of time. As long as the other guarantees holds, the application will not be able to disable the timer without the VM's permission.

To guard against malicious attacks as well as programming errors, we focus on the second type of approaches shown in Figure 3.5, where the node does not rely on the 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 [24]. This is unfortunate because the complexity of reducing the necessary run-time checks noted by native code systems like Harbor [41], is much lower when using a virtual machine. Compared to native CPU instruction sets, the

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	For each `INVOKELIGHT`, the total number of slots - the number of own variable slots for the caller >= the total variable slots for the callee.
T-9	At the point of an `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.
T-10	The stack is empty at branches and branch targets.
T-11	Before each instruction, the stack depth >= the number of elements to be consumed by the instruction.
T-12	After each instruction, the stack depth <= the max stack depth announced in the header.
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.

JVM instruction set is relatively simple and restricted, and easier to reason about. As we will see, many checks can be done at load time, reducing the need for runtime checks and the corresponding overhead.

For an interpreter, the VM is always in control, which makes it is easy to guarantee safety, but interpreters come at the cost of a one to two orders of magnitude performance penalty. Our VM is an Ahead-of-Time compiler, which translates JVM bytecode to native code at load-time to reduce interpretation overhead. This means the application is executing native code for most of the time, only occassionaly calling functions in the VM to handle more complex operations. However, all of this code is generated by the VM, which means it can perform checks at translation time to ensure the code it generates is safe. It only needs to insert run-time checks in cases where this is impossible to guarantee at translation time.

To guarantee safety we will first make the guarantees more concrete and specific to our VM:

- *control flow safety*: we are always executing
  - a translated JVM instruction from the top, so code can't jump to half way a generated instruction with undefined results, or
  - code in the VM itself, as a result of either a call to the VM from a translated JVM instruction, or returning from the main 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, or
  - the area of the heap assigned to the application

Both depend on the other: we will assume memory safety while we discuss control flow safety and vice versa. For each, our approach will be to first establish some gen-

eral constraints, and then examine each bytecode instruction to determine what additional checks are necessary to guarantee safety.

We will collect our checks in Table 6.1. Compared to the checks specified in Section 4.10 of the Java Virtual Machine Specification [48], the checks are different in two ways, they are specific to our VM's implementation, and since our goal is to ensure safety, our checks are less restrictive than the JVM specification. For example, they will allow writing to a private field from outside the class since this, while incorrect, is not a safety violation.

Many of our checks will depend on the *method header*. Each method in our VM has a small header defining properties such as the maximum stack size, number of local variables, return type, etc. Since the VM uses this header to determine the required size of the stack frame and the effects of a method call on the operand stack, many of the necessary checks are to ensure the actual bytecode follows the contract established in the method header. When the node receives code, it will first receive the headers for all methods, followed by the implementations, so the contracts for all methods are known when we start translating the byte code.

The first check we do is a basic sanity check on the data in the method headers (T-1). For example, since each parameter becomes a local variable, the number of local variable slots must be at least as high as the number of parameter, 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.

6.1 Control flow safety

We will first guarantee control flow safety by considering the effect of each possible JVM instruction and showing they either flow into the start of a legal next instruction, or return control back to the VM. To do so, we group them into four categories, shown in Table 6.2. Most instructions don't affect the control flow. The ones that do are: various kinds of branches, method invocations, and returns. 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 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
invokes	call a method, either through the VM or directly
returns	return to the address at the top of the stack
others	fall through to the next JVM 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, which is safe since these are part of the VM and will return to the same location.

For this category, the generated code will flow naturally into the next generated instruction, which is safe as long as there *is* a next instruction. This produces our second translation-time check, T-2: the last instruction in a method should be a `RETURN` or `GOTO` to prevent control from flowing into undefined territory after the method body.

6.1.2 Branch instructions

In our bytecode, which is a modified form of standard JVM bytecode, branches don't target an offset as in normal JVM bytecode, but a branch target ID. These targets are marked with a `BRTARGET` instruction, which doesn't emit any code, but causes 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 to handle forward branches.

This makes checking the branch addresses easy. Each method announces the number of branch targets that will be used in the method header. To ensure each taken branch will branch to a legal instruction within the method, we need to check that the target ID of each

branch is lower than the number of branch targets announced (T-3) so it points to an entry within the table, and that at the end of the method, all `BRTARGET` instructions have been found so all entries in the table point to the start of a translated instruction (T-4).

6.1.3 Invoke instructions

We have three kinds of method invocations in our VM.

For lightweight method calls, we require the implementation of the target method to come before any call to it, to ensure the address of target is known at translation time and we can directly generate a `CALL` to it. Ensuring this calls to a correct address is therefore trivial: we must simply check the code for the method has already 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, we generate a `CALL` to the VM's `callMethod` function, and pass it the id of the target method. At run time this id will be used as an index in the method table. Since we will store all the method headers before translating the implementations, we can check at translation time the method id is known (T-6), and since the VM won't start an application before the implementation for all methods is translated, this guarantees that `callMethod` will be able to find the correct target at runtime.

Finally, for `INVOKEVIRTUAL` and `INVOKEINTERFACE`, we do not know the target at translation time, since `callMethod` will resolve this depending on the object we will invoke the method on, which will not be known at runtime. Therefore we need a run-time check to ensure the method can be found (R-1). Since the method already needs to be resolved to make the call, this doesn't add any extra overhead.

6.1.4 Return instructions

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

The `RET` instruction will take the return address from the native stack, which is also used to store the JVM's integer operand stack. This means we need to check the integer operand stack is 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 any location.

The second way the return address could be corrupted is if the native stack overflows into the JVM heap. In our VM the heap is a fixed sized block that sits above other global variables, and the AVR's native stack grows down towards it. If the native stack were to grow into the area reserved for the heap, a return address may be corrupted.

We add a check during non-lightweight invokes that the stack frame for the called method, plus it's 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 stack to grow further. We can determine the maximum stack growth for calls our AOT generated code may make. Therefore we add a certain safety margin between the stack and 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 by the VM, stored in a number of global variables, and a block with static variables for each infusion. The infusion blocks are actually allocated on the heap at application startup time, but this part of the heap is then separated from the range the application may use to protect them from bad heap writes. 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 JVM stack frames containing space for local variables and reference operand stack, and the integer operand stack

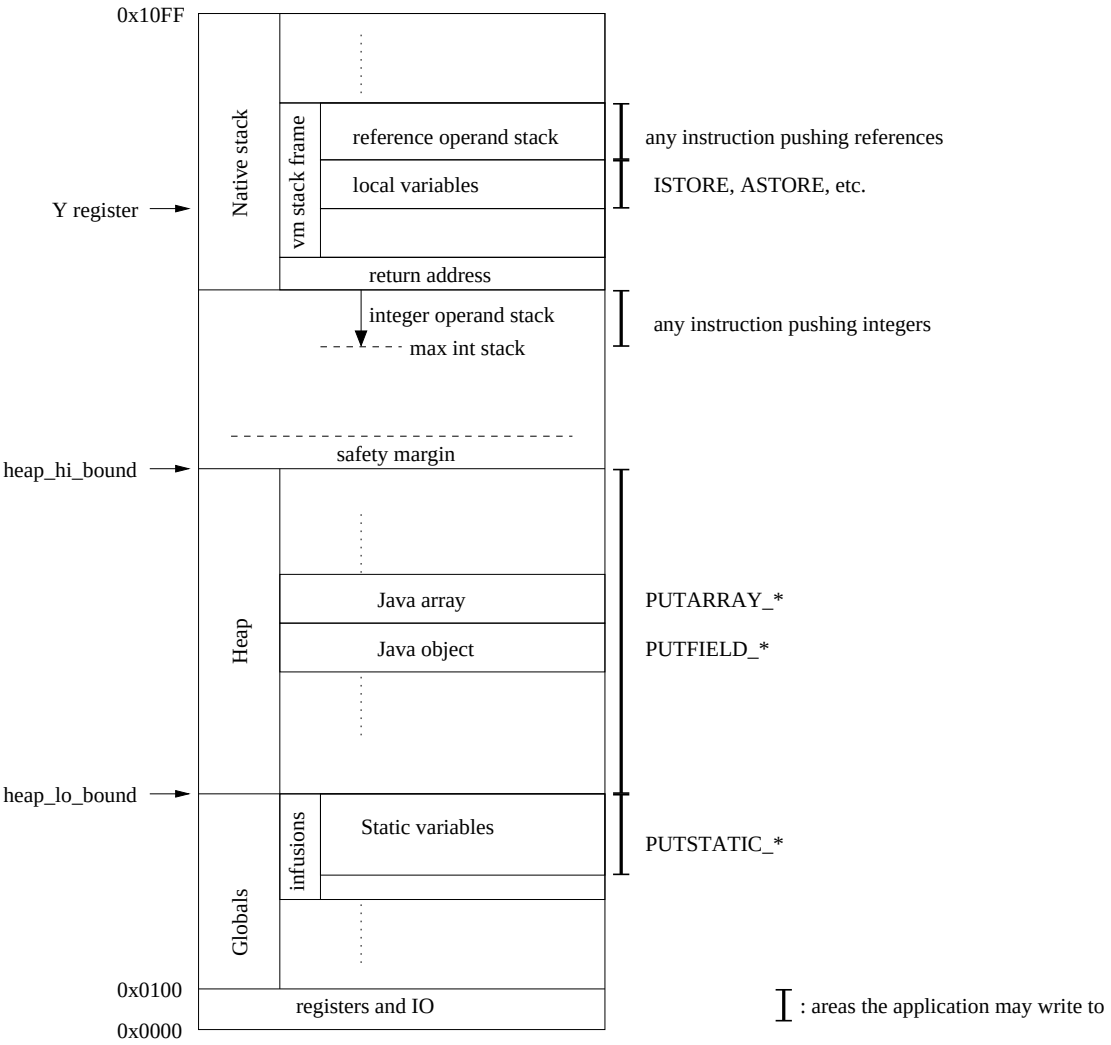


Figure 6.1: Global memory layout and the areas accessible to the application

which grows down directly on top of the native stack.

Thus, 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 our safety checks.

Having ensured control flow safety, we can now rely on the fact that we will always execute complete JVM instructions, and the application cannot skip past an inserted run-time check by jumping to the middle of a generated instruction. This means we can demonstrate memory safety by considering each how each JVM instruction writes to memory, and showing they are all either guaranteed to write to a correct address, or checked at run-time. Again, we group instructions in the categories shown below with respect to their writes to memory, as shown in Table 6.3, and consider each separately.

Table 6.3: Instructions writing to memory

type	writes to
`STORE`	local variable in stack frame
`PUTSTATIC`	static variable in infusion
`PUTARRAY`	heap
`PUTFIELD`	heap
any instruction pushing to the stack	reference or integer stack

The stack frame for a method contains space for its local variables and reference stack, as shown in Figure 6.1. For normal methods, the VM creates the stack frame based on the method header, so at load time we know how much space will be available at run-time. Lightweight methods don't create their own stack frame. Instead, they depend on the caller's stack frame, so whenever we translate an `INVOKELIGHT` instruction, we need to verify the stack frame of the current method has reserved enough free space for the lightweight method's locals and stack (T-8 and T-9). To do this, the method header contains both the total number of slots to reserve, and the number the method will use for itself.

Any remaining slots are used for lightweight methods.

6.2.1 The operand stack

The VM will reserve space for the operand stacks based on the information in the method header, so we need to make sure the actual stack depth neither underflows, or exceeds the maximum announced in the header.

The effect of each instruction on the stack is known at translation time. We simply the verification process by requiring the stack to be empty at all branches, so we can determine the stack depth in a single top to bottom pass. Alternatively, we could announce the expected stack state for each branch target in the method header so we can check the state matches at each branch and corresponding branchtarget. 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 four trivial rewrites to avoid the `? :` operator were necessary.

This allows us to verify the stack simply by maintaining two counters indicating how many values are on the integer and reference stacks, and updating these counters for each instruction's stack effects as we translate the method. 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 when translating these, the counters are initialised according to the number of arguments announced in the method header.

For each translated instruction, we then check there are enough values on the stack to consume its operands (T-11), and we do not exceed the maximum stack depth announced in the header after pushing it's results (T-12).

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, discussed below, require some more work to determine the stack effects.

Invoke instructions The `INVOKESTATIC`, `INVOKESPECIAL`, and `INVOKELIGHT` instructions all contain the id of the method that will be invoked. Since we already have

all method headers available at translation time, which contain the number of arguments and return type, it is easy to determine the stack effects of these invoke instructions.

For `INVOKEVIRTUAL` and `INVOKEINTERFACE` however, the actual method that will be called depends on the object on the stack at run-time. For these, we determine the stack effects based on the first method 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 try to send an implementation in a subclass that has different stack effects. We therefore add a run-time check (R-3) that will verify the method that is called at runtime, has the same stack effects as the one used to verify the stack at translation time.

Return instructions Note that `RETURN` instructions don't need any special care. The stack depth in the method is verified using the instruction found in the bytecode. It's 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 in registers to the calling method, 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 written to using the `STORE` instructions. Each `STORE` instruction contains the index of the local variable slot to write to, which makes it easy to verify at translation time.

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. The method header is used to create the current method's stack frame, or in the case of lightweight methods, to verify any caller has reserved enough slots for the lightweight method's locals.

Since we know how many slots will be available at the `Y` register, we only need to check at translation time that the index of the local is within the range announced in the method header to make sure we will write 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 a reference to an infusion, and the index of the static variable slot. At translation time, we simply need to check the referenced infusion exists (T-14), and the index is within the legal range (T-15) for that infusion.

6.2.4 `PUTFIELD` and `PUTARRAY`

The final type of memory access is to the heap. The various `NEW` instructions used to create arrays and objects are safe since they are fully implemented in the VM. Writes to heap memory happen using the `PUTFIELD` and `PUTARRAY` instructions, which write to object fields and array elements respectively.

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

In some cases it may be possible to verify of these operations at translation time, but this is hard without extensive analysis that would be too expensive for a sensor node. We therefore add a run-time check when translating these instructions just before doing the final memory access, to check the address is within the heap (R-4).

The VM will set 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 AVR's `Z` register. Just before the actual write to the heap, we insert al `CALL` to the `heapcheck` function shown in Listing 8.

This function checks the address in Z is within these bounds. If it is not, it will jump to the illegal_access_handler, allowing the VM to terminate the application. This check will add 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. This allows us to write 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 know at translation time, we simply load the object's address into Z and use STD to write to the correct offset.

This means the write could end up at most 63 bytes above the end of the heap. The cheapest way to avoid this would be to not use the STD instruction, but instead use the ADIW to add the offset to Z and use the normal ST instruction to store without displacement. However, this would add 2 bytes and 2 cycles for instruction. Instead we reuse the same small safety margin mentioned in check R-2. This is safe because we will never be writing to a heap object and executing 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, we can save 7 cycles by inlining in the check instead of calling it. However, this would increase the code size overhead from 4 bytes to over 30 bytes, which we consider too high.

If we align the top and bottom boundaries of the heap at 256 bytes, this would eliminate the need to bounds check the lower byte of the Z register, ZL. This would save 6 of the 22 cycles, but would waste 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, we can save 8 cycles if we keep the bounds in registers instead of memory, which would remove the need for the LDS instructions. However this reduces the performance of the stack cache since it means these registers are not available for stack caching. Which of these affects performance more depends on the code. We evaluate the difference in Section 7.7.1. Since the option to have the bounds in registers is more complex to

implement, thus increasing VM size, we choose to keep the bounds in registers.

```
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 8: Heap bounds check

6.3 Hardware acceleration

Sensor node CPUs do not have features such as memory management units (MMUs) and privileged-mode execution used in desktop CPUs. While the added complexity for these cannot be justified in a market driven by extreme pressure to minimise chip cost and area, some more lightweight support from the hardware could significantly reduce the cost of providing safe execution environments, with or without a VM. In this section we consider two possible extensions to the CPU that can be implemented at little cost in terms of chip surface area, and we believe to be generic enough to justify their inclusion in these CPUs.

The two most expensive checks are R-2 and R-4. The last is the cause of most of the run-time overhead, and both require us to keep a safety margin in between the heap and stack. While this margin is small, RAM memory is a scarce resource we would prefer not to waste.

We propose two checks to be added to the CPU, causing it to trap to a fixed error handler when the check fails to allow the node to terminate the program.

6.3.1 Stack overflow protection

To protect against stack overflow, we propose extending the CPU with a stack bound register. The CPU compares the stack pointer to this stack bound whenever it is modified, either by a direct write or implicitly by a `PUSH` instruction. If the stack pointer drops below the stack bound, the CPU traps to the error handler.

In our VM, the stack bound register would be set to the value of `heap_hi_bound`, thus eliminating the need for R-2. This is also be a useful safety measure to have for any other embedded code since the low amount of RAM makes stack overflows a real risk in any application, which can lead to unpredictable behaviour and hard to diagnose bugs.

6.3.2 Heap write protection

To eliminate the need to call `heapcheck` before each write, we propose adding two more registers containing the lower and upper bounds of the area the application is legally allowed to write to. The CPU then checks that writes to memory fall within these bounds, otherwise it traps to the error handler.

The question then becomes which writes should be checked, since the VM should still be able to write to any location, and CPU is not aware of which instructions are part of the VM or part of the application. Several options may be considered here.

Checked store instructions The AVR's instruction set contains several blocks of unused opcodes [8]. There is enough space to add checked versions of the 'store indirect with displacement' instruction through register `Z`, which we use to access the heap. When a store happens using these checked versions, the CPU does the bounds checks, while stores using the unchecked versions can still write to anywhere in memory.

Our VM could then use these checked versions when generating code for `PUTFIELD` and `PUTARRAY` to implement R-4 at no runtime cost, and use the unchecked versions for stores that are known to be safe at load-time, such as `PUTSTATIC`. This would also be useful for native code approaches like Harbor, which normally verifies writes to memory are guarded by a check similar to ours. Instead it may now allow direct stores as well, as

long as the checked store instructions are used, and avoid the run-time overhead in these cases.

However, using the majority of the currently unused opcodes for this feature may be too high a price to pay.

Protected mode Alternatively, we may add a status bit to the CPU to indicate a protected mode, and use only two unused opcodes for instructions to set or clear this protected mode bit. When the CPU is in protected mode, all stores are required to fall within the heap bounds, while in unprotected mode the entire address range is accessible.

Instead emitting a call to `heapcheck`, our VM may now protect the stores for `PUTFIELD` and `PUTARRAY` by setting and unsetting the protected mode bit. Since these instructions should only take a single cycle, this would reduce the cost of such checks from 22 to just 2 cycles. Similar to the previous case, native code approaches can also benefit by allowing both writes to memory through the existing guard methods, as well as direct stores preceded by the instruction to turn on protected mode.

Chapter 7

Evaluation

We use a set of eleven different benchmarks to measure the effect of our optimisations:

- *bubble sort*: taken from the Darjeeling sources, and used in [13, 21]
- *heap sort*: standard heap sort [3]
- *binary search*: taken from the TakaTuka [7] source code
- *fft*: fixed point FFT, adapted from the widespread fix_fft.c and used in [41]
- *outlier detection*: our implementation of the algorithm described in [41]
- *xxtea*: as published in [86]
- *md5*: also taken from the Darjeeling sources, and used in [13, 21]
- *rc5*: from LibTomCrypt [46]
- *CoreMark 1.0*: a freely available benchmark developed by EEMBC [16]
- *MoteTrack*: RSSI based mote tracking [50, 51]
- *Heat detection*: adapted from code used in our group to track objects using an 8x8 pixel heat sensor

The first seven are small benchmarks, consisting of only one or two methods. They all process an array of data, which we expect to be common on a sensor node, and likely to be

a performance sensitive operation. However, the processing they do is different for each benchmark, allowing us to examine how our optimisations respond to different kinds of code. The eighth benchmark, CoreMark, is a standard benchmark representative of larger embedded applications.

For each benchmark we implemented both a C and a Java version, keeping both implementations as close as possible. We manually optimised the code as described in Section 5.2. These optimisations did not affect the performance of the C version, indicating `avr-gcc` already does similar transformations on the original code. We use `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.

We manually examined the compiled code produced by `avr-gcc`. While we identified some points where more efficient code could have been generated, except for the constant shifts mentioned in the previous section, this did not affect performance by more than a few percent. This leads us to believe `avr-gcc` is a fair benchmark to compare to.

We run our VM in the cycle-accurate Avrora simulator [78], emulating an ATmega128 processor. We modified Avrora to get detailed traces of the compilation process and of the run-time performance of both C and AOT compiled code.

Our main measurement for both code size and performance is the overhead compared to optimised native C. To compare different benchmarks, we normalise this overhead 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.

7.1 Benchmarks

In this section we will briefly describe how the source code for each benchmark was obtained, and describe any relevant details in their implementation.

7.1.1 MoteTrack

7.2 CoreMark

First, we will examine the CoreMark benchmark. CoreMark 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

As mentioned before, we kept the Java versions as close to the original C code as possible. The other benchmarks are all relatively simple, and porting them to Java is straightforward. CoreMark is a much more comprehensive benchmark, and the more complex code exposes some challenges when using Java on embedded devices.

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 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 `core_list_mergesort`.

Finally, 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 `list_head` and `list_data` structs, shown in Listing 9. 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 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 initialise the list. This is also the faster option

Table 7.1: Effect of manual source optimisation on the CoreMark benchmark

	list		matrix		state		total	
	time [a]	vs nat. C	time	vs nat. C	time	vs nat. C	time	vs nat. C
native C	17.9		49.8		18.4		86.0	
baseline	122.0	(+583%)	367.8	(+639%)	293.7	(+1496%)	783.5	(+811%)
optimised, using original source	52.6	(+195%)	239.1	(+380%)	82.8	(+350%)	374.4	(+335%)
manually inline small methods	-0.4	(-3%)	-37.2	(-75%)	-17.0	(-92%)	-54.5	(-63%)
use short array index variables	+6.8	(+39%)	-109.4	(-219%)	-5.0	(-27%)	-107.6	(-125%)
avoid recalculating expressions in a loop	0.0	(-1%)	-7.6	(-15%)	0.0	(0%)	-7.6	(-9%)
reduce array and object access	-0.2	(-1%)	-18.1	(-37%)	-2.4	(-14%)	-20.7	(-24%)
reduce branch cost in crcu8	-4.0	(-22%)	-0.5	(-1%)	-3.5	(-19%)	-8.1	(-10%)
using optimised source	54.8	(+207%)	66.3	(+33%)	54.9	(+198%)	175.9	(+104%)
(unfair) avoid creating objects	-3.4	(-19%)	0.0	(0%)	-11.4	(-61%)	-14.7	(-17%)
(unfair) avoid virtual calls	-22.8	(-128%)	0.0	(0%)	0.0	(0%)	-22.8	(-26%)
after 'unfair' optimisations	28.6	(+60%)	66.3	(+33%)	43.5	(+137%)	138.4	(+61%)

[a] in millions of cycles

because accessing 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 9: C and Java version of the CoreMark list data structures

7.2.1 Manual optimisations

After translating the C to Java code, we only do 'fair' manual optimisations that we believe a future optimising infuser could easily do automatically. Since CoreMark is our most comprehensive benchmark, we use it to evaluate the effect of these manual optimisations.

Table 7.1 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 using any of our optimisations, is 811% slower than native C. Even after applying all our other optimisations, the best we can achieve with the original code is a 335% slowdown, proving the importance of a better optimising infuser.

Next we apply our manual optimisations to the Java source code, as described in Sec-

tion 5.2, and add a small extra optimisation to `crcu8` which can be easily reorganised to reduce branch overhead.

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. The reason the linked list part is slightly slower after using short array index variables is that it allocates a small object, and the change in memory usage means this now triggers a run of the garbage collector, which presumably had already happened earlier in the version with int index variables. Combined these optimisations reduce the overhead for the whole benchmark from 335% to 104%.

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

In the rest of the evaluation, all the results presented are for the manually optimised Java code.

7.2.2 'Unfair' optimisations

After these optimisations, CoreMark is still the slowest of our benchmarks, and the only one to still be at more than 100% overhead. We can improve performance further if we relax our constraint of only doing optimisations that a compiler could do automatically without changing the code significantly.

In Table 7.1 we see 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 we see all three parts taking roughly the same time. The state machine and linked list processing both suffer from a much larger slowdown than the matrix part, which by itself would be the second 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.

The linked list benchmark also creates a small object, but here the biggest source of

overhead is in the virtual method call to the compare objects in `core_list_mergesort` that we use instead of a function pointer. Virtual methods cannot be made lightweight.

This is the best we can do when we strictly translate the C to Java code, and only do optimisations that could be done automatically. If we relax this constraint, we can remove these two sources of overhead as well: because we know we the code will not run multithreaded or recurse, we could choose to statically allocate the small objects used by the state machine, and one by the linked list part, since they only use 90 bytes. The virtual call to the comparer objects in the list benchmark is the most natural implementation this in Java, but given that we know 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 61% overhead over native C, right in the middle of the spectrum of the other benchmarks. However, the code is now fundamentally different than the original CoreMark, so it is not a completely fair comparison, although a developer writing this code in Java from the start may have made similar choices.

Either way, 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 we used here cannot be applied.

Neither of these two optimisations were used in the rest of the evaluation.

7.3 AOT translation overhead

Next we will look at the effect of our different optimisations for the baseline AOT translation approach, for all eight of 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 then

Table 7.2: Performance data per benchmark

BENCHMARK	b.sort	h.sort	b.srch	fft	xxtea	md5	rc5	coremk	average
EXECUTED JVM INSTRUCTIONS (%)									
Load/Store	79.8	72.1	58.8	57.8	50.9	43.7	41.1	55.5	57.5
Constant load	0.2	8.1	9.8	10.8	12.5	19.1	17.6	10.1	11.0
Processing	8.0	7.8	13.1	22.8	32.4	28.9	36.6	14.0	20.5
math	8.0	5.6	9.2	12.0	10.1	12.5	10.7	8.3	9.6
bit shift	0.0	2.3	3.9	7.5	8.1	5.4	8.0	2.2	4.7
bit logic	0.0	0.0	0.0	3.2	14.2	11.0	17.9	3.6	6.2
Branches	12.0	11.0	17.6	4.1	4.0	5.8	2.3	16.0	9.1
Invoke	0.0	0.5	0.0	0.0	0.0	0.0	0.0	0.4	0.1
Others	0.0	0.5	0.7	4.5	0.2	2.5	2.4	4.0	1.9
STACK									
Max. stack (bytes)	8	8	8	8	24	20	14	18	13.5
Avg. stack (bytes)	2.6	2.9	2.8	3.0	11.8	6.3	6.8	3.2	4.9
PERFORMANCE OVERHEAD BEFORE OPTIMISATIONS (%)									
Total	496.7	351.4	430.8	522.6	251.5	226.5	123.4	359.0	345.2
push/pop	183.5	139.4	192.5	220.2	168.0	105.9	61.3	128.2	149.9
load/store	200.1	144.3	180.9	132.7	42.5	43.9	28.5	91.9	108.1
mov(w)	10.4	2.9	-1.2	6.2	2.4	1.7	-1.7	4.0	3.1
other	102.7	64.8	58.7	163.5	38.6	75.0	35.3	134.9	84.2
PERFORMANCE OVERHEAD REDUCTION PER OPTIMISATION (%)									
Impr. peephole	-162.8	-118.8	-116.3	-99.9	-61.8	-51.7	-23.2	-61.0	-86.9
Stack caching	-22.9	-29.5	-76.8	-129.8	-97.3	-54.3	-38.6	-40.0	-61.1
Pop. val. caching	-116.6	-73.3	-29.8	-52.4	-6.9	-12.9	-8.8	-26.0	-40.9
Mark loops	-62.3	-28.8	-84.4	-40.2	+5.1	-10.9	-8.1	-40.4	-33.7
Const shift	0.0	-9.2	-22.4	-80.4	-18.5	-43.8	-20.2	-10.2	-25.6
16-bit array index	-37.5	-25.3	-36.5	-22.4	-13.8	-5.5	-4.1	-39.3	-23.1
SIMUL	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-36.7	-4.6
PERFORMANCE OVERHEAD AFTER OPTIMISATIONS (%)									
Total	94.6	66.5	64.6	97.5	58.3	47.4	20.4	105.4	69.3
push/pop	0.0	-7.0	0.0	0.0	37.4	0.1	2.9	5.1	4.8
load/store	9.0	33.2	28.2	22.5	-2.3	20.3	4.3	17.6	16.6
mov(w)	10.4	3.9	10.8	4.6	5.6	2.2	0.5	10.0	6.0
other	75.3	36.4	25.6	70.4	17.6	24.8	12.7	72.7	41.9

group them into 4 categories that roughly match the 3 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. Before we introduce stack caching, it emits moves to replace push/pop pairs, and after the mark loops 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 by 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 7.2 and 7.3.

In Figure 7.1 we see how our optimisations combine to reduce performance overhead. We take the average of the 8 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. We start with the original AOT approach with only the simple peephole optimiser, and then incrementally add each of our optimisations. The lightweight method call optimisation is already included in these results. Its effect will be examined in detail in Section 7.6.

Using the simple optimiser, the types 1, 2 and 3 overhead are all significant, at 150%, 108%, and 84% respectively. The basic approach does not have many reasons to emit a move, so we see that in some cases the AOT version actually spends fewer cycles on move instructions than the C version, resulting in small negative values. When we improve the

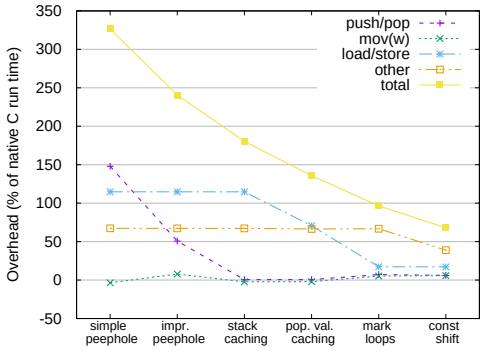


Figure 7.1: Perf. overhead per category

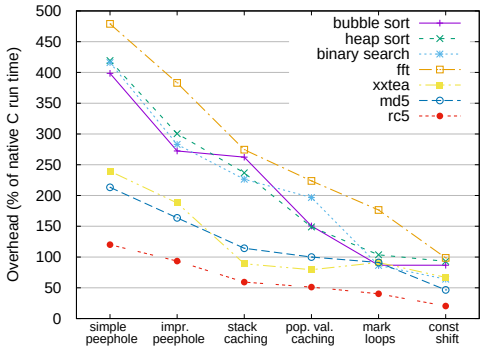


Figure 7.2: Perf. overhead per benchmark

peephole optimiser to include non-consecutive push/pop pairs, push/pop overhead drops by 98.1% (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 11.7% in move overhead. For a 16-bit value 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 11.7% extra move overhead matches a 93.6% reduction in push/pop overhead.

Next we introduce stack caching 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, we now add popped value caching 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 we have fewer registers available for stack caching means we have to spill stack values to memory more often. While we save 46% on loads and stores, the push/pop and move overhead both increase by 6%.

Most of the push/pop and load/store overhead has now been eliminated and the type

Table 7.3: Code size data per benchmark

BENCHMARK	b.sort	h.sort	b.srch	fft	xxtea	md5	rc5	coremk	average
CODE SIZE (BYTES)									
JVM	78	140	91	493	384	2986	457	5719	
Native C	150	416	212	1214	1442	9458	910	10388	
AOT original	520	1170	616	2694	3780	29362	4074	33668	
AOT optimised	344	738	450	1460	2268	14798	2140	25560	
CODE SIZE OVERHEAD BEFORE OPTIMISATIONS (%)									
Total	242.1	179.9	190.6	121.9	162.1	210.4	347.7	223.8	209.8
push/pop	57.9	61.2	52.8	55.7	102.6	133.1	163.1	74.5	87.6
load/store	89.5	64.1	69.8	31.8	28.4	56.7	67.9	53.5	57.7
mov(w)	1.3	1.4	0.9	0.3	0.7	-2.7	-1.3	1.4	0.3
other	93.4	53.1	67.0	34.1	30.4	23.3	118.0	94.4	64.2
CODE SIZE OVERHEAD REDUCTION PER OPTIMISATION (%)									
Impr. peephole	-57.9	-41.1	-45.3	-26.5	-38.5	-54.3	-62.4	-30.7	-44.6
Stack caching	-13.1	-20.6	-24.5	-37.1	-56.1	-78.6	-106.4	-18.7	-44.4
Pop. val. caching	-18.5	-27.8	0.0	-13.8	-6.2	-18.8	-17.8	-12.6	-14.4
Mark loops	-2.6	+4.8	+7.5	-5.9	+6.0	-1.1	-3.7	-3.8	0.2
Const shift	0.0	-2.4	-4.7	-5.3	+1.6	+4.0	-5.5	-1.1	-1.7
16-bit array index	-23.7	-16.2	-11.3	-13.0	-11.6	-5.1	-16.7	-8.8	-13.3
SIMUL	0.0	0.0	0.0	0.0	0.0	0.0	0.0	-2.3	-0.3
CODE SIZE OVERHEAD AFTER OPTIMISATIONS (%)									
Total	126.3	76.6	112.3	20.3	57.3	56.5	135.2	145.8	91.3
push/pop	21.1	5.7	7.5	0.0	13.3	0.0	4.4	19.4	8.9
load/store	31.6	33.5	47.2	4.1	14.8	37.2	25.3	36.8	28.8
mov(w)	0.0	3.8	4.7	-2.6	2.5	-1.8	17.8	5.4	3.7
other	73.7	33.5	52.8	18.8	26.6	21.0	87.7	84.2	49.8

3 overhead, unaffected by these optimisations, has become the most significant source of overhead. This type has many different causes, but we can eliminate half of it with our three instruction set optimisations. These optimisations, especially the 16-bit array index, also reduce register pressure, so we also see slight decreases in the other overhead types, although this is minimal in comparison. The CoreMark benchmark is the only one to do 16-bit to 32-bit multiplication, so the average performance improvement for `SIMUL` is small, but Table 7.2 shows it is very significant for CoreMark.

Combined, these optimisations reduce performance overhead from 345% to 69% of native C performance.

7.4 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

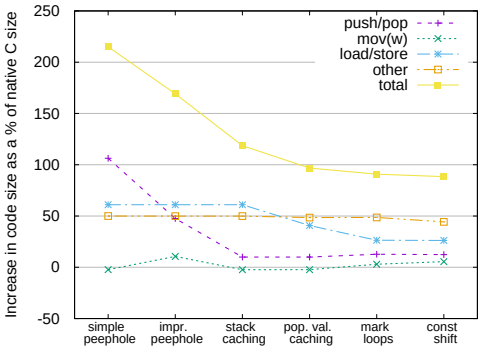


Figure 7.3: Code size overhead per category

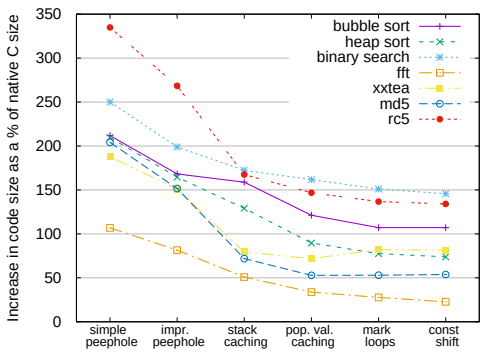


Figure 7.4: Code size overhead per benchmark

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 34%, but the effect on code size varies per benchmark: some are slightly smaller, others slightly larger.

For each value that is live at the beginning of the loop, we need to emit the load before the mark loop block, so in terms of code size we only benefit if it is loaded more than once, and may actually lose some if it is then popped destructively, since we would need to emit a `mov`. Stores follow a similar argument. Also, for small methods the extra registers used may mean we have to save more call-saved registers 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. 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 will be smaller, but md5 actually shows a small 4% increase in code size since it contains many shifts by a large number of bits.

Using 16-bit array indexes also reduces code size. The benchmarks here already have the manual code optimisations, so they use short index variables. This means the infuser will emit a `S2I` instruction 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 about 13% code size overhead in addition to the 23% reduction in performance overhead. Using 32-bit variables in the source code would also remove the need for `S2I` instructions, but the extra code to manipulate the 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 7.4 shows the difference in code size between the AOT translator and Darjeeling's interpreter. The basic AOT approach is 6245B larger than the interpreter, and each of our optimisations adds a little to the size of the VM.

They also generate significantly smaller code. The second 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 278 bytes to the VM, but it reduces the size of the generated code by 14.4%. If we have more than 1.9KB 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 18.1KB.

As is often the case, there is a tradeoff between size and performance. The interpreter is smaller than each version of our AOT compiler, and Table 7.3 shows JVM 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 device.

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. Using only popped

Table 7.4: Code size and memory consumption

	size vs interpreter	size vs baseline		AOT code reduction	break even	memory usage
Baseline	6245 B					30 B
Improved peephole	6523 B	278 B	(+278)	-14.4%	1.9 KB	30 B
Simple stack caching	7243 B	998 B	(+720)	-28.6%	3.5 KB	41 B
Popped value caching	8607 B	2362 B	(+1364)	-33.3%	7.1 KB	89 B
Markloop	11903 B	5658 B	(+3296)	-33.2%	17.0 KB	98 B
Const shift	12373 B	6128 B	(+470)	-33.8%	18.1 KB	98 B
16-bit array index	12353 B	6108 B	(-20)	-38.0%	16.1 KB	98 B
SIMUL	12419 B	6174 B	(+66)	-38.1%	16.2 KB	98 B
Lightweight methods	12961 B	6716 B	(+542)	-38.4%	17.5 KB	98 B

value stack caching reduces code size by 33.3%, and results in a performance overhead of 156%. The 16-bit array index optimisation should also be included, since this reduces the size of both the VM and the generated code.

7.4.2 VM memory consumption

The last column in Table 7.4 shows the size of the main data structure that needs to be kept in memory while translating a method. For the baseline approach we only use 30 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 word 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 98 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 7.2 shows the distribution of the JVM instructions executed in each benchmark,

and both the maximum and average number of bytes on the JVM stack. We can see some important differences between the benchmarks. While the benchmarks on the left are almost completely load/store bounded, towards the right the benchmarks become more computation intensive, spending fewer instructions on loads and stores, and more on math or bitwise operations. The left benchmarks 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 7.2 and 7.3 first shows the overhead before optimisation, split up in the four 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 sort and search, leaving less overhead to remove for stack caching. Moving to the right, the more complicated expressions mean there is more distance between a push and a pop, leaving more cases that cannot be handled by the peephole optimiser, and replacing it 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 on the right, the load/store overhead is much smaller, but the higher stack size means stack caching is very important for these benchmarks.

The first seven benchmarks are smaller benchmarks that can highlight certain specific aspects of our approach, the CoreMark benchmark represents larger sensor node applications, and is a mix of different types of processing. As a result, it is an average case in almost every row in Table 7.2. The reason it ends up being the slowest after all optimisations was discussed in 7.2.2. With the 'unfair' optimisations described there, CoreMark's performance overhead would be 61%, very close to the average of the other benchmarks.

Bit shifts Interestingly, the reason fft is the slowest, is similar to the reason rc5 is fastest: they both spend a large amount of time doing bit shifts. Rc5 shifts by a variable, but large number of bits. Only 8.0% of the executed JVM instructions are bit shifts, but they account for 71% of the execution time in the optimised version. For these variable bit shifts, our translator and `avr-gcc` generate a similar loop, so the two share a large constant factor.

On the other hand fft is a hard case because it does many constant shifts by exactly 6 bits. For these, our VM simply emits 6 single shifts, which is slower than the special case `avr-gcc` emits for shifts by exactly 6 bits. While we could do the same, we feel this special case is too specific to include in our VM.

Bubble sort Next we look at bubble sort in some more detail. After optimisation, we see most of the stack related overhead has been eliminated and of the 94.6% remaining performance overhead, most is due to other sources. For bubble sort there is a single, clearly identifiable source. When we examine the detailed trace output, this overhead is largely 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 seven additions for an array of ints. This calculation is repeated for each access, while the C version has a much more efficient approach, using the auto-increment version of the AVR's LD and ST instructions to slide a pointer over the array. Of the remaining 94.6% overhead, 73% is caused by these address calculations.

Xxtea and the mark loops optimisation Perhaps the most interesting benchmark is xxtea. Its high average stack depth means popped value caching does not have much effect: most registers are used for real stack 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 5.1%, and code size overhead increases 6%! Here we have an interesting tradeoff: 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

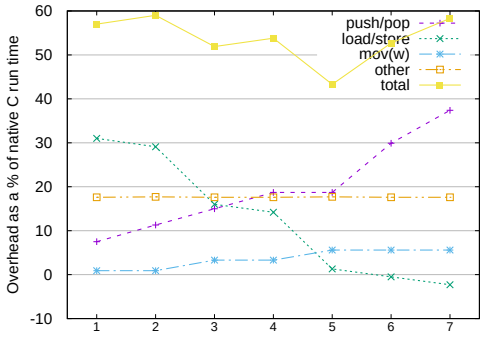


Figure 7.5: Xxtea performance overhead for different number of pinned register pairs

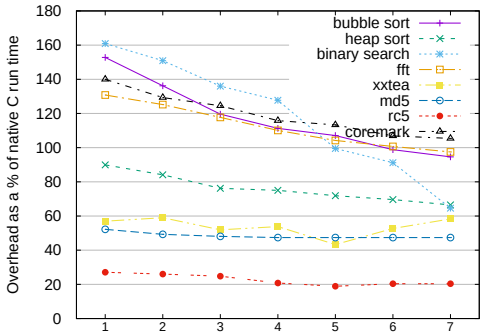


Figure 7.6: Per benchmark performance overhead different number of pinned register pairs

memory.

For most benchmarks the maximum of 7 register pairs to pin variables to 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 the 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. 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 actually 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.

Figure 7.6 shows the performance for each benchmark, as the number of pinned register pairs is increased. The three benchmarks that stay stable or even slow down when the number pinned pairs is increased beyond 5 are exactly the benchmarks that have a high stack depth: xxtea, md5 and rc5. 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

Table 7.5: Methods per benchmark and relative performance for normal, lightweight invocation, and inlining. Highlights indicate changes from the versions used to obtain the results in the previous sections.

	# calls	C	Java Base version	Java Alternative version	Java Using normal method calls
CoreMark					
ee_isdigit	3920	normal (inlined)	manually inlined	lightweight (JVM)	manually inlined
core_state_transition	1024	normal	lightweight	lightweight	normal
crcu8	584	normal (inlined)	lightweight	lightweight	normal
crcu16	292	normal	lightweight	lightweight	normal
calc_func	220	normal	lightweight	lightweight	normal
compare_idx	209	normal (inlined)	normal (virtual)	normal (virtual)	normal (virtual)
core_list_find	206	normal	lightweight	lightweight	normal
compare_complex	110	normal	normal (virtual)	normal (virtual)	normal (virtual)
crcu32	64	normal	lightweight	lightweight	normal
matrix_sum	16	normal	lightweight	lightweight	normal
others (<16 calls each)	39	normal	normal	normal	normal
cycles			3482185	3639967	5030231
overhead v native C			105.4%	114.7%	196.7%
code size			25560	25576	26282
FFT					
FIX_MPY	768	marked inline	manually inlined	lightweight (JVM)	normal
SIN8	63	marked inline	manually inlined	ProGuard inlined	ProGuard inlined
COS8	63	marked inline	manually inlined	ProGuard inlined	ProGuard inlined
cycles			78241	113611	562650
overhead v native C			97.5%	186.8%	1320.4%
code size			1460	1410	1530
heap sort					
SWAP	1642	#define	manually inlined	manually inlined	manually inlined
siftDown	383	normal	lightweight	manually inlined	normal
cycles			289845	286749	563967
overhead v native C			66.5%	64.7%	223.9%
code size			738	926	758

current VM always pins 7 pairs, we used this as our end result and leave this heuristic to future work.

7.6 Method invocation

Most of our benchmarks consist of only a single method. The three small functions in the FFT benchmark were inlined by the C compiler, so we manually inlined them in the Java version. Heap sort does contain a real method call: it consist of a main loop, repeatedly calling the `siftDown` method. CoreMark is a much more extensive benchmark consisting of many methods.

In this section we will examine the effect of the lightweight method calls on these three

benchmarks, compared to inlined code and normal method calls.

In Table 7.5 we see the most frequently called methods 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 contains 3 methods 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 smallest functions that were inlined by the C compiler. The most commonly called methods were transformed to lightweight methods, simply by adding the `@Lightweight` annotation where possible. For Java versions of the `compare_idx` and `compare_complex` methods, this was not possible since we do not support lightweight virtual methods.

In the next two columns we vary these choices slightly to examine the effect of our 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 coded the lightweight method to use only the stack and no local variables. This slowed the benchmark down by 4.5%, adding 157,782 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.

Here we see 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 again 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`, have to be implemented as normal

Java methods. This adds a total of 1,548,046 cycles, making it almost 1.5 times slower than the lightweight methods version. Spread over 2406 calls, this means the average method invocation added over 643 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 894 function calls. In the C and normal Java versions these are inlined. When we change them all to normal Java methods, ProGuard will automatically inline the `SIN8` and `COS8` methods, adding only a minimal overhead, but the `FIX_MPY` method is too large for ProGuard to inline. If we mark it `@Lightweight` the large number of calls relative to the total running time means the average overhead of over 40 cycles per invocation slows down the benchmark by 45%. Without lightweight methods, this would be as high as 619%

Finally, for the heap sort benchmark we normally use a lightweight method for `sift-Down`. In the second version we see that, like in the CoreMark example, the difference between inlining and the lightweight method is small: we only gain 1% by manual inlining. However, the benchmark runs almost twice as long when we use a normal method call instead of a lightweight method. A significant increase, but less than FFT since heap sort does half as many calls and has a higher total running time to spread the call overhead.

In terms of code size, we can see normal methods take slightly more space than a lightweight method. Listing 5 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 smaller than the call, but the heap sort benchmark shows that inlining larger methods can result in significantly larger code.

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 7.6: Cost of safety guarantees

BENCHMARK	b.sort	h.sort	b.srch	fft	xxtea	md5	rc5	coremk	average
PERFORMANCE OVERHEAD VS NATIVE C (%)									
unsafe	94.4	66.2	64.3	96.7	57.6	45.7	19.5	110	69.3
safe writes	185.9	107.7	64.3	145.6	68.2	60.3	22.2	137.3	98.9
safe reads and writes	277.4	192.6	119.1	201.5	100	80.3	33.4	217.9	152.8
PERFORMANCE OVERHEAD VS UNSAFE VM (%)									
safe writes	47.1	25	0	24.9	6.7	10	2.3	13	17.5
safe reads and writes	94.1	76.1	33.4	53.3	26.9	23.7	11.6	51.4	49.3
CODE SIZE OVERHEAD VS NATIVE C (%)									
unsafe	102.6	67.9	85.8	17.6	55.1	54.9	120.7	50.1	69.3
safe writes	107.9	71.8	85.8	20.3	56.2	55.7	124.2	54	72
safe reads and writes	113.2	77.5	89.6	24.2	60.1	59.1	131.2	61.3	77
CODE SIZE OVERHEAD VS UNSAFE VM (%)									
safe writes	2.6	2.3	0	2.3	0.7	0.5	1.6	2.6	1.6
safe reads and writes	5.2	5.7	2	5.6	3.2	2.7	4.8	7.5	4.5

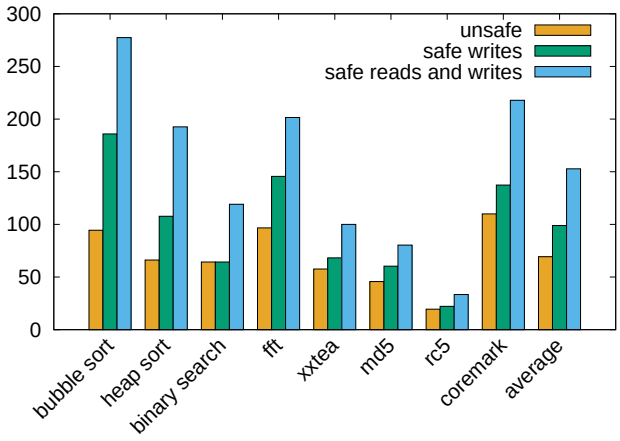


Figure 7.7: Overhead increase due to safety checks

7.7 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 load-time. This leads to both a very modest increase in VM complexity due to the safety checks, and a lower run-time overhead. In this section we evaluate both.

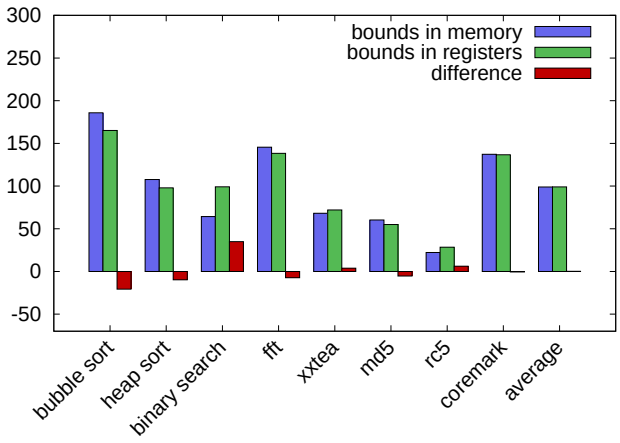


Figure 7.8: Comparison of safety cost with bounds in memory or registers

7.7.1 Run-time cost

First we examine the run-time cost of our safety guarantees. We use 8 benchmarks, 7 smaller ones and the larger CoreMark benchmark [16] which consists of several functions, implementing a state machine, linked list processing, and several matrix operations. All benchmarks are implemented in both C and Java, with the Java implementation following the original C code as closely as possible. We then compare the running times for both versions and report the overhead, presented as a percentage of the optimised native C runtime, i.e. a 100% overhead corresponds to a run time for the Java version that is twice as long as the corresponding C implementation.

Table 7.6 shows the results for our 8 benchmarks. Figure 7.7 shows the data in the first section of the table. The baseline is the unsafe version of our VM, which is on average 69% slower than native C. When we add our safety checks this overhead increases to 98%, corresponding to an 18% increase in runtime compared to the unsafe VM.

We can see the cost of safety depends greatly on the benchmark we run. Most checks are done at load-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 object field or array write is within the heap bounds.

Thus, the run-time overhead is determined by the number of object or array writes a benchmark does. Bubble sort has the highest increase in running time as a result of these checks at 47%, while binary search, which does no writes at all is unaffected. CoreMark,

being a large benchmark with a mix of operations, is close to the average slowdown.

Safe reads Next we consider read safety. Up to this point our VM only checks the application cannot *write* to memory it's 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 5-30% [**?**].

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 obviously needs to be protected. However, in many sensor node applications the node may only be running a single application, and our VM does not contain any state that would be useful to an attacker. In these cases, only providing write safety will be sufficient.

Adding read safety to our VM is trivial: instructions to read from local and static variables is already protected since these instructions reuse the same code to access them as the instructions to write to them. For heap access, we simply add the same call to `heapcheck` to the `GETARRAY` and `GETFIELD` just before the actual read.

Looking at Figure 7.7, we see 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 roughly 50% on average, corresponding to a 152% slowdown over native C. The per benchmark picture is similar to that of write-only safety with bubble sort, which spends almost all it's time reading and writing from and to arrays, being the worst hit, while rc5 is still only 33% slower than C. Again, CoreMark performs very close to the overall average.

Keeping heap bounds in registers In Section 6.2.4 we considered several alternatives to our chosen heap bounds check, one of which was to keep the bounds in dedicated registers to avoid having to fetch them from memory for each check. Here we evaluate

this choice.

The expected performance for this alternative is easily calculated. Having the bounds in registers would reduce the cost of the check from 22 to 14 cycles, reducing the safety overhead by 36%. However, we would have 4 registers less to use for stack caching. We ran our experiments a second time for our unsafe VM, this time reducing the number of registers available to the stack cache by 4. Since this doesn't affect the number of heap accesses, we then added the observed overhead for safety checks, reduced by 36%.

Figure 7.8 shows the overhead for our chosen approach with the heap bounds in memory, compared to the expected overhead is we keep the heap bounds in registers. For some benchmarks such as bubble sort, the savings in heap bounds checks outweighs the reduced effectiveness of the stack cache, But the improvement in performance is rather small at 20%, and other for benchmarks the reverse is true, with binary search overhead increased by 35%. Overall the benchmarks were nearly perfectly balanced on average, as was 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 tradeoff, in this case VM size and complexity, and our calculations show that even choosing the best option for each of the smaller benchmarks only reduces overhead by a few percent.

7.7.2 Code-size cost

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

Most of our checks are not more complex than comparing two integers, and failing if a condition is not met. The most complex part is deciding the stack effects of instructions to guard against stack over- or underflow. 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 1776 bytes.

As we can see in Table 7.6, the size of the code the VM generates increases by only 1.6%. Since most checks occur at load-time, most instructions produce exactly the same native code in the safe version of our VM. The exceptions are `INVOKEVIRTUAL` and `INVOKEINTERFACE`, which now contains the expected stack effects to realise check R-3, and the array and object write instructions `PUTFIELD` and `PUTARRAY`, which now emit a single extra `CALL` instruction the `heapcheck` routine. Since these instructions are both relatively rare, and already generate a larger block of native instructions than most instructions do in the unsafe version, the total effect on code size is very limited.

7.7.3 Comparison to native code alternatives

As discussed in Section 3.4, several non-VM approaches exist to guarantee safety on a sensor node. Two of these, *t-kernel* and Harbor, allow the node to guarantee safety independent of the host. In this section we compare these to our approach, 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 range as our VM. However both *t-kernel* and our approach provide additional advantages. In *t-kernel*'s case a form of virtual memory, and for our 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 larger at a 6-8.5x increase, limiting the size of programmes we can load onto the device.

A better comparison is possible for Harbor, which only provides safety. The overhead reported is in the range of 160 to 1230%. The latter is for a synthetic benchmark, filling a block of memory with arbitrary data. The authors note this simulates a very common behaviour of sensor network applications: copying sampled sensor data into a buffer that can be transmitted into the network.

We implemented this as a benchmark that fills an array of 256 elements with an arbitrary number. This is one of the worst cases for our VM since consecutive array writes are expensive for two reasons: (i) in our VM this results in repeated executions of the `PUTARRAY` instruction, which calculates the target address for each write, while native code can

Table 7.7: Overhead for the fill array benchmark as a percentage of native C

Benchmark	Unsafe overhead	Safe overhead	Increase due to safety checks
8-bit bytes	210%	566%	356%
16-bit shorts	182%	452%	270%
32-bit ints	155%	336%	181%

slide a pointer over the array, eliminating the need for repeated address calculations. (ii) each of these writes will trigger a call to `heapcheck`.

Our benchmark is implemented as a simple loop as shown in Listing 10.

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

Listing 10: Fill array benchmark (8-bit version)

The resulting overhead is shown in Table 7.7. We implemented our benchmark for arrays of bytes, shorts and ints. Unfortunately the Harbor paper does not mention the size of elements in their buffer, nor whether this will make a difference for their overhead. For our VM, the worst case is filling a byte array because here the relative overhead from address calculation and safety checks is highest.

The worst case overhead due to safety checks is 356%, considerably less than Harbor's 1230%, and dropping to 181% when storing ints instead of bytes. Our VM also incurs other overhead, not related to safety checks, but the total overhead of 566% in the worst case is still less than half of Harbor's.

While our VM is currently faster than Harbor, the comparison is not entirely fair. Harbor lists the cycle overhead for all of it's 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, our `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 our VM is more coarse. If Harbor could be modified to use a check similar to ours, it's overhead could potentially be reduced to $1230/65 * 22 \approx 416\%$. This puts it in the middle of our three versions in terms

Table 7.8: Comparison of our approach to related work

Approach	Platform independent	Safe	Performance	Code size
Native code	No	No	1x	1x
Interpreters	Yes	Mostly no	300-23000% slower	50% smaller
Ellul's AOT	Yes	No	123-844% slower	120-346% larger
Safe TinyOS	No	Yes[a]	17% slower	27% larger
t-kernel	No	Yes	50-200% slower	500-750% larger
Harbor	No	Yes	160-1230% slower	30-65% larger
Our VM (unsafe)	Yes	No	20-110% slower	17-120% larger
Our VM (safe)	Yes	Yes	22-185% slower	20-124% larger

[a] but requires a trusted host

of 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 segment.

7.8 Worst case benchmarks

to do

7.9 Limitations and the cost of using a VM

- recursive - threads - lots of tiny objects - no multidimensional arrays

summarise cost of vm in terms of code size, performance, memory usage, reduced flexibility Refer back to Suganuma, who on page two mentions Java code typically results in many short method calls. This is bad for us.

7.10 Conclusion

Since we consider both safety and platform independence to be desirable properties for sensor networks, we conclude our evaluation by comparing our approach to existing work on both sensor node virtual machines and safety in Table 7.8.

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. t-kernel and Harbor provide safety independent of the host, but at the cost of a significant increase in code size, or performance overhead respectively. Non of these approaches provide platform independence.

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

Coming back to the question of whether a VM is a good way to provide security, we first note that since to the best of our knowledge only two such systems exists, we cannot exclude the possibility that native code approaches could be further optimised to achieve better performance.

However our results show that currently our approach is on-par with or faster than the two existing native code approaches, and provides platform independence at the same time. We also note that if we require a platform independent way of reprogramming our nodes, making it safe comes at a relatively low extra cost, with our safe VM only 18% slower than the unsafe version.

Chapter 8

Lessons from JVM

While developing MyVM, we encountered a number of situations where Java/JVM was not a good match for sensor node code. In this chapter we will discuss these.

In Section 1.3 we defined our main research 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. The main motivation to base MyVM on Java was the availability of a rich set of tools and infrastructure to build on, including a solid VM to start from in the form of Darjeeling.

One aspect of Java/JVM that makes it an attractive choice for sensor nodes is its simplicity, allowing (a subset of) it to be implement in as little as 8KB [32]. However, it also lacks some features that make it a less good fit for a typical sensor node code in a number of situations. These range from lack of type definitions and explicit inlining that are minor annoyances that reduce code readability, to the lack of support for constant data that could be a show stopped for a number of applications.

Many of the points we raise could be improved with minor changes to Java, leading us to a 'sensor node Java', much like nesC [27] is a sensor node version of C, but some require more drastic changes. We believe if we were to develop a sensor node VM from scratch, with the aim of providing platform independence, safety, and performance through AOT compilation, we would end up with a design very different from JVM.

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. While we haven't

Table 8.1: Point requiring attention in future sensor node VMs

Sec.	Issue	in	affects
8.1	A tailored standard library	Standard library	VM size
8.2	Support for constant data	Source language, JVM	memory usage, application siz
8.3	Support for nested data structures	Source language, JVM	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, JVM	(predictable) performance
8.9	Reconsidering adv. language features OO, GC, threads, exceptions	Source language, JVM	VM size, complexity, and performance

Table 8.2: 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

implemented any of these changes, and the possible solutions we suggest require much more study to be turned into a working system, we believe this list will be valuable to future sensor node VM developers.

8.1 A tailored standard library

A minimum Java APIs for resource constrained devices was proposed by Sun Microsystems, namely the Connected Limited Device Configuration (CLDC) specification [59]. 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 program space for features that are rarely required for sensor node applications. Table 8.2 shows the code size of library support as implemented in the Darjeeling VM as used in the WuKong

project [66].

The largest mismatch comes from the CLDC's string support, which takes up over 8KB. 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 typically don't have a UI but 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.

We argue that a tailored 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.

8.2 Support for constant data

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

This is especially important for arrays of constant data, which are common in WSN applications. When we implement this as a `final` Java array, the compiler emits a static class initialiser that creates a Java array object, and then uses the normal array access instructions to initialise each element individually:

There are two problems with this: (i) the array will occupy scarce RAM; and (ii) initialising array elements using JVM instructions requires 4 instructions per element, resulting in 1669 bytes of code to initialise a 256 byte array.

Arrays of constant data appear in several of our benchmarks. The LEC benchmark contains two arrays of 17 16-bit and 8-bit, which are small enough to store in RAM. In this case the lack of support for constant data wastes a few bytes of memory, but doesn't prevent us from implementing the algorithm. However, in 3 other cases it did.

CoreMark The CoreMark benchmark contains a number of constant arrays with the expected outcome of the benchmark. This data is not used in the actual measured benchmark, but only to verify its results are correct. When storing this as normal Java arrays, they took up too much memory to run the benchmark, we transformed them into functions that return the correct value depending on an index passed as a parameter.

FFT The FFT benchmark contains an array of sine wave values that would be too expensive to calculate at run-time. For the 8 bit version, this contains 256 bytes, which is small enough to fit in RAM. However, the 16-bit version contained an array of 1024 16-bit values, which in itself would still fit in memory, but did not leave enough free memory run the benchmark.

In this case this was solved by storing this array in flash and adding a Java method implemented in C to read from it. This allows us to run the benchmark, but means we would only be able to run the benchmark on devices with this specific function available.

MoteTrack Finally, the MoteTrack benchmark contains a large dictionary of reference RSSI signatures that are used to determine a node's location by matching the observed RSSI values to known locations in the reference database.

At 20560 bytes, this dictionary is small enough to fit in flash memory, but over 5 times larger than the available RAM. Similar to the 16-bit FFT benchmark, the only way to implement this in a VM without support for this kind of constant data, is to store it in flash and use a native method to read the signatures.

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

8.3 Support for nested data structures

Besides the need to support constant data, the MoteTrack benchmark exposes another weakness of Java: it does not support complex data structures of many small elements (anything larger than a primitive type) efficiently. Listing 11 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.

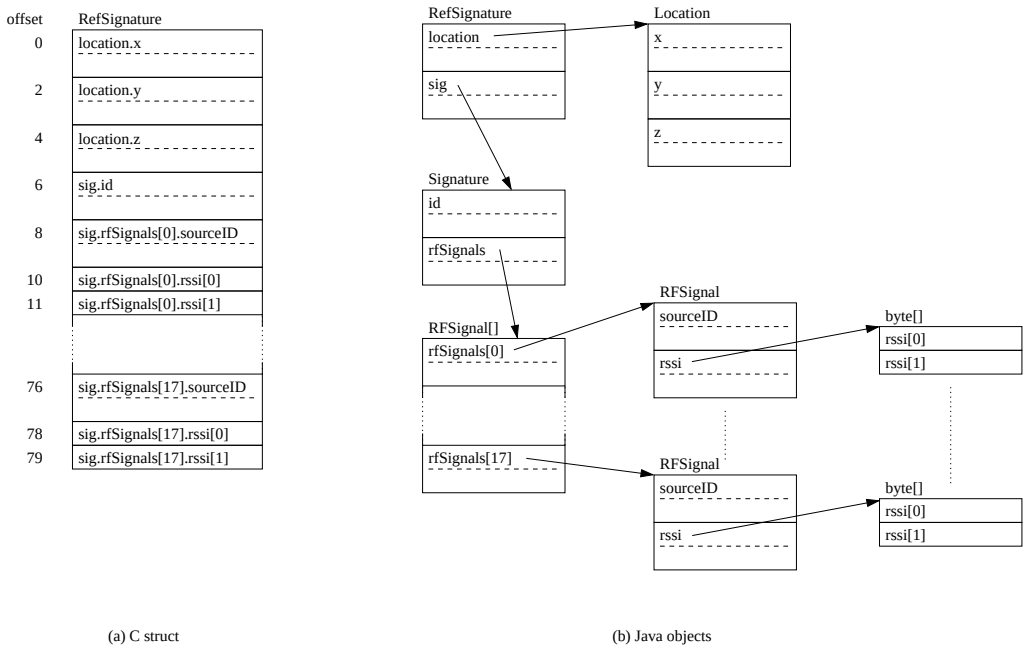
```
#define NBR_RFSIGNALS_IN_SIGNATURE 18
#define NBR_FREQCHANNELS            2

struct RefSignature
{
    Point location;
    Signature sig;
};

struct Point
{
    uint16_t x;
    uint16_t y;
    uint16_t z;
};

struct Signature
{
    uint16_t id;
    RFSignal rfSignals[NBR_RFSIGNALS_IN_SIGNATURE];
};

struct RFSignal
{
    uint16_t sourceID;
    uint8_t rssi[NBR_FREQCHANNELS];
};
```

Listing 11: 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.

In Java, we have no concept of a struct. 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. Thus, the most natural way to translate the C structures in Listing 11 to Java, is as a collection of objects 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.

There are two problems with this. First, since the location of these Java objects is not known until runtime, there is a performance penalty for having to follow the chain of references. MoteTrack will loop over the signals to compare two signatures. If we start from the `rfSignals` array, Java needs to lookup the right element in `rfSignals` to



(a) C struct

(b) Java objects

Figure 8.1: The `RefSignature` data structure as C struct, and collection of Java objects

get the right signal, then lookup the `rssi` field to get the RSSI array, and then another array lookup to get the right 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 offsets.

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 Darjeeling. In addition, the 19 arrays have a 3 byte header, and the Java objects and arrays contain a total of 39 references, which take up 2 bytes each. In total, the collection of Java objects takes up:

$80 + 40 * 5 + 19 * 3 + 39 * 2 = 415$ bytes

Combined with MoteTrack's other data structures, this became too large to fit in memory, which forced us to refactor the 2 element `rssi` array into two byte variable stored directly in `RFSignal`, as explained in Section 7.1.1. This allowed us to run the benchmark, but a RefSignature still takes up 235 bytes, and reading a rssi value still takes two lookups.

Generally, arrays of primitive types don't suffer from this problem and can be stored

efficiently, but large arrays of objects cause significant overhead. In many cases we can work around this problem by flattening the structure, for instance in MoteTrack's case we could replace the array of `RFSignals` with three separate arrays for `sourceIDs`, `rssi_0` and `rssi_1`, but only at a significant cost in readability.

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 (instance, local or static) and stack values are stored as 32-bit, even if the actual type is shorter. On a sensor node this wastes memory, and causes a performance overhead since most nodes have 8-bit or 16-bit architectures, so many sensor node JVMs, including Darjeeling, introduce 16-bit operations and store values in 16-bit slots.

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

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

```
a=(short)(b+c);
```

Passing literal integer values to a method call treats them as ints, even if they are short enough to fit in a smaller type, which means we end up with 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.

Possible solutions If we want to have an efficient sensor node VM, both from a performance and memory usage perspective, better support for data types smaller than 32-bit integers is necessary.

We suggest that C-style automatic narrowing conversions would make most sensor

node code more readable, but to leave the option of Java's default behaviour open, we may implement this as new datatypes: Declaring variable `a` as `unchecked short` would implictly narrow to short when needed, so `a=b+c;` would not need an explicit cast anymore, while declaring it as a normal `short` would.

8.5 Simple type definitions

When developing code for a sensor node, the limited resources force us to adopt 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. But 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 we had 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 also 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 VM does not have the resources for this. We found this often is a problem for small helper functions that are frequently called. As an example, a C version of the xxtea cipher [86] contains this macro:

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

This macro is called in four places, and is very performance critical. Tools like Proguard [72] can be used to inline small methods, but in this case it is larger than Proguard's size threshold. This leaves developers with two unattractive options: either leaving it as a method and accepting the performance penalty, or manually copy-pasting the code, which is error-prone and leads to code that is harder to maintain.

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

Another option is to give the developer more control over inlining, which could easily be achieved by adding an `inline` keyword to force the compiler to inline important methods. These annotations are usually placed at the method level. As an extension, since not all calls may be equally important for performance, it may be useful allow the developer to save code space by placing the `inline` keyword at a call instead of at the method level to only inline specific, performance sensitive calls.

8.7 An optimising compiler

As discussed in previous sections, 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 runtime behaviour. However, a sensor node does not have the resources for this and must execute the code as it is received. This leads to significant overhead, for example by repeatedly reevaluating a constant expression in a loop.

Possible solutions Even without a clear performance model, some basic optimisations can be done. In the experiments described in Section 7.2.1, we described some very con-

servative optimisations that already result in code twice as fast as the original.

8.8 Allocating objects on stack

While memory management is too big a topic to cover completely, we do want to mention one relevant observation. In Java anything larger than a primitive value has to be allocated on the heap. This introduces a performance overhead, both for allocating the objects, and the occasional garbage collection run, which may take several thousand cycles.

As an example, one function in the CoreMark benchmark uses two small local arrays of 8 ints, which are on the stack in C, but need to be on the heap in Java. In code that frequently needs short-lived objects this overhead can be significant, and unpredictable GC runs are a problem for code with specific timing constraints. In CoreMark's case this adds up to a performance overhead of about 60% relative to C.

Possible solutions We suspect it may be possible to avoid this cost in many cases by extending the VM's memory model to allow us to allocate objects and arrays on the stack under certain conditions. We know from TakaTuka's experience that there are many cases where we can determine at compile time that an object can be garbage collected at a certain point [6].

If the compiler can determine an object can be freed at the end of the method in which it was created, we can allocate space for it in the method's stack frame. This avoids the overhead of allocating on the heap, and the occasional garbage collection run triggered by this.

8.9 Reconsidering advanced language features

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

While features like threads and garbage collection are all useful, they come at a cost.

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

In Table 8.2 we show the code size for some features we discuss below. These were determined by counting the size of functions 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 VM.

These features also cause a performance penalty, which is significant when Ahead-of-Time (AOT) compilation is used, and features such as threads and exceptions are much harder in an AOT compiler where we can't implement them in the interpreter loop. This means that if we care about performance and the corresponding reduction in CPU energy consumption, we either have to give them up, or spend considerably more in terms of VM complexity and size.

8.9.1 Threads

As shown in Table 8.2, support for threads costs about 10% of the VM size, if we exclude the string library. In addition, there is the question of allocating a stack for each thread. If we allocate a fixed block, it must be large enough to avoid stack overflows, but too large a block wastes precious RAM. Darjeeling allocates each stack as a linked list of frames on the heap. This is memory efficient, but allocating on the heap is slower and occasionally triggers the GC.

We therefore argue a more cooperative concurrency model is more appropriate where lightweight threads voluntarily yield the CPU and share a single stack.

8.9.2 Exceptions

In terms of code size, exceptions are not very expensive to implement in an interpreter, but again, they are harder to implement in an AOT compiler. We also feel the advantage of

having exceptions is much lower than the other features mentioned in this section. They could be easily replaced with return values to signal errors.

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 most likely less than 2KB, but the overhead for resolving a virtual method call is considerable, and an AOT compiler can generate much more efficient code for static calls.

In practice we seldom used virtual methods in sensor node code, but some form of indirect calls is necessary for things like signal handling. It should be possible to develop a 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 one to change. While the first three features could be changed with minor modifications to Java, the managed heap is at its very core.

Still, there are good reasons for considering alternatives. Table 8.2 shows the GC methods in Darjeeling add up to about 3.5KB, but the actual cost is much higher as many other parts of Darjeeling are influenced by the garbage collector.

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

8.10 Conclusion and future work

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 we present suggests many applications will be affected by at least some.

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

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

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

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

Chapter 9

Conclusion

A major problem for sensor node VMs has been performance. Most interpreters are between one to two orders of magnitude slower than native code, leading to both lower maximum throughput and increased energy consumption.

Previous work on AOT translation to native code by Ellul and Martinez [22] improves performance, but still a significant overhead remains, and the tradeoff is that the resulting native code takes up much more space, limiting the size of programmes that can be loaded onto a device. For the CoreMark benchmark, the performance is 9x slower than native C, and the code 3.5 times larger.

In this paper, we presented the complete set of techniques we developed to mitigate this code size overhead and to further improve performance. We evaluated their effectiveness using a set of benchmarks, some with specific characteristics to highlight the results in more extreme conditions, and include the larger CoreMark benchmark to represent the average behaviour of larger sensor node applications. Combined, our optimisations result in a compiler that produces code that is on average only 1.7 times slower and 1.9 times larger than optimised C.

These optimisations do increase the size of our 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. This leads us to believe that these optimisations will be useful in many scenarios, and make using a VM a viable option for a wider range of applications.

Many opportunities for future work remain. In this paper we focus on techniques for the sensor node side, but a future VM should come with a better optimising infuser on the host to prepare better quality bytecode. This infuser should also support inlining small methods as efficiently as manual inlining, and in most cases automatically determine which methods should be made lightweight.

For the mark loops optimisation, a heuristic is needed to make a better decision on the number of registers to pin, and we can consider applying this optimisation to other blocks that have a single point of entry and exit as well. Since supporting preemptive threads is expensive to implement without the interpreter loop as a place to switch threads, we believe a cooperative concurrency model where threads explicitly yield control is more suitable for sensor nodes using AOT, and we are working on building this on top of Darjeeling's existing thread support.

A more general question is what the most suitable architecture and instruction set is for a VM on tiny devices. Hsieh et al. note that the performance problem lies in the mismatch between the VM and the native machine architecture [34]. In this paper we presented a number of modifications to the bytecode format to make it better suited for use on a sensor now, but ultimately we believe JVM is not the best choice for a sensor node VM. It has some advanced features, such as exceptions, preemptive threads, and garbage collection, which add complexity but may not be necessary on a tiny device. At the same time, there is no support for constant data, which is common in embedded code: a table with sine wave values in the fft benchmark is represented as a normal array at run-time, using up valuable memory. We may also consider extending the bytecode with instructions to express common operations more efficiently. For example, an instruction to loop over an array such as the one found in Lua [36] would allow us to generate more efficient code and eliminate most of the remaining overhead in the bubble sort benchmark.

Our reason to use JVM is the availability of a lot of infrastructure to build on. Like Hsieh et al., we do not claim that Java is the best answer for a sensor node VM, but we believe the techniques presented here will be useful in developing better sensor node VMs, regardless of the exact instruction set used.

One important question that should be considered is whether that instruction set should be stack-based or register-based. Many modern bytecode formats are register-based, and a number of publications report on the advantages of this approach [89, 70]. However, these tradeoffs are quite different for a powerful JIT compiler, and a resource-constrained VM. When working with tiny devices, an important advantage of a stack-based architecture is its simplicity, and our results here show that much of the overhead associated with the stack-based approach can be eliminated during the translation process.

Bibliography

[1] B. Alpern et al. Implementing jalapeño in java. In *OOPSLA*, 1999.

[2] Anon. Python-on-a-chip, 2011.

[3] Anon. Heap sort at rosetta code, 2015.

[4] 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.

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

[6] F. Aslam, L. Fennell, C. Schindelhauer, P. Thiemann, G. Ernst, E. Haussmann, S. Rührup, and Z. A. Uzmi. Optimized Java Binary and Virtual Machine for Tiny Motes. In *DCOSS*, 2010.

[7] F. Aslam et al. Introducing takatuka - a java virtualmachine for motes. In *SENSYS*, 2008.

[8] Atmel. AVR Instruction Set Manual. 1997.

[9] Atmel. 8-bit Atmel Microcontroller with 128KBytes In-System Programmable Flash ATmega128 ATmega128L. 2011.

[10] R. Balani et al. Multi-level software reconfiguration for sensor networks. In *EMSOFT*, 2006.

[11] K. C. Barr and K. Asanović. Energy-Aware Lossless Data Compression. *ACM Transactions on Computer Systems (TOCS)*, 24(3), Aug. 2006.

[12] D. Braginsky and D. Estrin. Rumor Routing Algorithm For Sensor Networks. *WSNA '02 Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 22–31, Sept. 2002.

[13] N. Brouwers, K. Langendoen, and P. Corke. Darjeeling, a feature-rich vm for the resource poor. In *SENSYS*, 2009.

[14] M. Chang and P. Bonnet. Meeting ecologists' requirements with adaptive data acquisition. *SenSys '10 Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, pages 141–154, 2010.

[15] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent Types for Low-Level Programming. In *Proceeding ESOP'07 Proceedings of the 16th European Symposium on Programming*, pages 520–535, Mar. 2007.

[16] T. E. M. B. Consortium. Coremark 1.0, 2017.

[17] N. Cooprider, W. Archer, E. Eide, and D. Gay. Efficient memory safety for TinyOS. In *Sensys '07*, 2007.

[18] A. Courbot, G. Grimaud, and J.-J. Vandewalle. Efficient off-board deployment and customization of virtual machine-based embedded systems. *ACM Trans. Embed. Comput. Syst.*, 9(3), 2 2010.

[19] 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 8th ACM Conference on Embedded Networked Sensor Systems*, 2010.

[20] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and C. Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. *OOPSLA '96 Proceedings of the 11th*

ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 83–100, 1996.

[21] J. Ellul. *Run-time Compilation Techniques for Wireless Sensor Networks*. PhD thesis, University of Southampton, 2012.

[22] J. Ellul and K. Martinez. Run-time compilation of bytecode in sensor networks. In *SENSORCOMM*, 2010.

[23] M. A. Ertl. Stack caching for interpreters. In *PLDI*, 1995.

[24] L. Evers. *Concise and Flexible Programming of Wireless Sensor Networks*. PhD thesis, University of Twente, 2010.

[25] C.-L. Fok, G.-C. Roman, and C. Lu. Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications. *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICSCS'05)*, 2005.

[26] A. Gal, C. W. Probst, and M. Franz. Hotpathvm: An effective jit compiler for resource-constrained devices. In *VEE*, 2006.

[27] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language:A Holistic Approach to Networked Embedded Systems. *Proceedings of Programming Language Design and Implementation (PLDI)*, June 2003.

[28] D. George. Micropython, 2017.

[29] J. Gosling et al. *The Java Language Specification, Java SE 8 Edition*. 2015.

[30] L. Gu and J. A. Stankovic. t-kernel: providing reliable OS support to wireless sensor networks. In *SenSys '06*, 2006.

[31] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A Dynamic Operating System for Sensor Nodes. *MobiSys '05: The Third International Conference on Mobile Systems, Applications, and Services*, pages 163–176, May 2005.

[32] T. Harbaum. Nanovm, June 2006.

[33] K. Hong et al. Tinyvm, an efficient virtual machine infrastructure for sensor networks. In *SENSYS*, 2009.

[34] C.-H. A. Hsieh, J. C. Gyllenhaal, and W.-m. W. Hwu. Java bytecode to native code translation: The caffeine prototype and preliminary results. In *MICRO*, 1996.

[35] 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. In *2012 IEEE 11th International Symposium on Network Computing and Applications (NCA)*, 2012.

[36] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The implementation of lua 5.0. *Journal of Universal Computer Science*, 11(7), 2005.

[37] C. Intanogonwiwat, R. Govindan, and D. Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. *MobiCom '00 Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 56–67, Aug. 2000.

[38] 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 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 294–310, Oct. 2000.

[39] J. Koshy and R. Pandey. Vm*: Synthesizing scalable runtime environments for sensor networks. In *SENSYS*, 2005.

[40] A. Krall. Efficient javavm just-in-time compilation. In *PACT*, 1998.

[41] R. Kumar, E. Kohler, and M. Srivastava. Harbor: software-based memory protection for sensor nodes. In *IPSN '07*, 2007.

[42] K. Langendoen, A. Baggio, and O. Visser. Murphy Loves Potatoes: Experiences from a Pilot Sensor Network Deployment in Precision Agriculture. *14th Int. Work-*

shop on Parallel and Distributed Real-Time Systems (WPDRTS) (apr 2006), pages 1–8, Mar. 2006.

[43] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *ASPLOS*, 2002.

[44] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *NSDI*, 2005.

[45] 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*, Nov. 2004.

[46] T. libtom. Libtomcrypt, 2017.

[47] K.-J. Lin, N. Reijers, Y.-C. Wang, C.-S. Shih, and J. Y. Hsu. Building Smart M2M Applications Using the WuKong Profile Framework. In *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, pages 1175–1180, 2013.

[48] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. The Java® Virtual Machine Specification Java SE 9 Edition. pages 1–618, Aug. 2017.

[49] 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 3rd international conference on Mobile systems, applications, and services*, 2005.

[50] K. Lorincz and M. Welsh. Motetrack: A robust, decentralized approach to rf-based location tracking, 2006.

[51] K. Lorincz and M. Welsh. MoteTrack: a robust, decentralized approach to RF-based location tracking. *Springer Personal and Ubiquitous Computing Special Issue on Location and Context-Awareness*, 11(6):489–503, Oct. 2006.

[52] K. Lorincz et al. Mercury: A wearable sensor network platform for high-fidelity motion analysis. In *SENSYS*, 2009.

[53] 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 (TODS)*, 30(1), 2005.

[54] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless Sensor Networks for Habitat Monitoring. *WSNA '02 Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, Aug. 2002.

[55] F. Marcelloni and M. Vecchio. An Efficient Lossless Compression Algorithm for Tiny Nodes of Monitoring Wireless Sensor Networks. *The Computer Journal*, 52(8):969–987, Nov. 2009.

[56] R. Müller, G. Alonso, and D. Kossmann. A virtual machine for sensor networks. In *EuroSys*, 2007.

[57] G. Muller et al. Harissa: A flexible and efficient java environment mixing bytecode and compiled code. In *COOTS*, 1997.

[58] D. Niculescu and B. Nath. Ad hoc positioning system (APS). In *IEEE Global Telecommunications Conference GLOBECOM '01*, pages 2926–2931. IEEE, Nov. 2001.

[59] Oracle. Connected limited device configuration, 2005.

[60] 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.

[61] K. Pister. On the Limits and Applications of MEMS Sensor Networks. Technical report, UC Berkeley, Feb. 2001.

[62] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling Ultra-Low Power Wireless Research. *IPSN '05 Proceedings of the 4th international symposium on Information processing in sensor networks*, Apr. 2005.

[63] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson. Toba: Java For ApplicationsA Way Ahead of Time (WAT) Compiler. *COOTS'97 Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies*, pages 1–13, June 1997.

[64] 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*, pages 33–49, Feb. 2014.

[65] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *WSNA '03 Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 60–67, 2003.

[66] 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)*, 2013.

[67] C. Savarese, J. Rabaey, and K. Langendoen. Robust Positioning Algorithms for Distributed Ad-Hoc Wireless Sensor Networks. *ATEC '02 Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 317–327, Apr. 2002.

[68] 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 1st ACM international workshop on Wireless sensor networks and applications*, pages 112–121, Aug. 2002.

[69] N. Shaylor, D. N. Simon, and W. R. Bush. A Java Virtual Machine Architecture for Very Small Devices. *LCTES '03 Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, June 2003.

[70] Y. Shi et al. Virtual machine showdown: Stack versus registers. In *VEE*, 2005.

[71] 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. *Proceedings of the 2nd international conference on Virtual execution environments (VEE)*, 2006.

[72] G. Square. Proguard 5.3.2, 2016.

[73] 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, 2014*, 2014.

[74] C. Technology. MICAz Wireless Measurement System datasheet.

[75] Texas Instruments, Incorporated. MSP430x1xx Family User's Guide (slau049f). 2006.

[76] Texas Instruments, Incorporated. MSP430F15x, MSP430F16x, MSP430F161x MIXED SIGNAL MICROCONTROLLER (SLAS368G). 2011.

[77] B. L. Titzer. Virgil: Objects on the Head of a Pin. *OOPSLA '06 Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2006.

[78] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *IPSN*, 2005.

[79] P. Tyma. Why are we using Java again? *Communications of the ACM*, 41:38–42, June 1998.

[80] T. van Dam and K. Langendoen. An Adaptive Energy-Efficient MAC Protocol for Wireless Sensor Networks. *SenSys '03 Proceedings of the 1st international conference on Embedded networked sensor systems*, Jan. 2018.

[81] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP '93*, 1993.

[82] 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. In *2014 IEEE International Conference on Internet of Things (iThings), and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom*, pages 148–154, 2014.

[83] B. Warneke, M. Last, B. Liebowitz, and K. S. J. Pister. Smart Dust:Communicatingwith a Cubic-Millimeter Computer. *IEEE Computer*, 34:44–51, Jan. 2001.

[84] M. Weiser. The Computer for the 21st Century. *Scientific American*, pages 94–104, Sept. 1991.

[85] 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 7th symposium on Operating systems design and implementation*, pages 381–396, Nov. 2006.

[86] D. Wheeler and R. Needham. Xxtea: Correction to xtea. Technical report, Computer Laboratory, University of Cambridge, 1998.

[87] I. Wirjawan et al. Balancing computation and communication costs: The case for hybrid execution in sensor networks. *Elsevier Ad Hoc Networks*, 6(8), Nov. 2008.

[88] W. Ye, J. Heidemann, and D. Estrin. An Energy-Efficient MAC Protocol for WirelessSensor Networks. *NFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies.*, Nov. 2002.

[89] Y. Zhang et al. Swift: A register-based jit compiler for embedded jvms. *ACM SIGPLAN Notices*, 47(7), jan 2012.