Improved Ahead-of-time Compilation of Stack-based JVM Bytecode on Resource-constrained Devices

NIELS REIJERS and CHI-SHENG SHIH, NTU-IoX Research Center

Many virtual machines exist for sensor nodes with only a few kB RAM and tens- to a few hundred kB Flash memory. They pack an impressive set of features but suffer from a slowdown of one to two orders of magnitude compared to optimised native code, reducing throughput and increasing power consumption.

Compiling bytecode to native code to improve performance has been studied extensively for larger devices, but the restricted resources on sensor nodes mean most modern techniques cannot be applied. Simply replacing bytecode instructions with predefined sequences of native instructions is known to improve performance but produces code several times larger than the optimised C equivalent, limiting the size of programmes that can fit onto a device.

This article identifies the major sources of overhead resulting from this basic approach and presents optimisations to remove most of the remaining performance overhead, and over half the size overhead, reducing them to 67% and 77%, respectively. While this increases the size of the VM, the break-even point at which this fixed cost is compensated for by the smaller code it generates, is well within the range of memory available on a sensor device, allowing us to both improve performance and load more code on a device.

CCS Concepts: • Computer systems organization → Sensor networks; Embedded software; Sensors and actuators; • Software and its engineering → Virtual machines; Compilers;

Additional Key Words and Phrases: Wireless sensor networks, resource-constrained, JVM, bytecode, virtual machines, compilers, ahead-of-time compilation

ACM Reference format:
Niels Reijers and Chi-Sheng Shih. 2019. Improved Ahead-of-time Compilation of Stack-based JVM Bytecode on Resource-constrained Devices. ACM Trans. Sen. Netw. 15, 3, Article 34 (August 2019), 44 pages. https://doi.org/10.1145/3341170

1 INTRODUCTION

Internet-of-Things 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: powerful enough to run Linux, but relatively expensive and power hungry. On the other end are CPUs like the Atmel ATmega or TI MSP430, commonly used in sensor nodes: much

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.
Part of this work was previously published as “Ahead-of-Time Compilation of Stack-Based JVM Bytecode on Resource-Constrained Devices,” by Niels Reijers and Chi-Sheng Shih, at the International Conference on Embedded Wireless Systems and Networks (EWSN) 2017.
Authors’ address: N. Reijers and C.-S. Shih, NTU-IoX Research Center, Department of Computer Science and Information Engineering, National Taiwan University, No. 1, Section 4, Roosevelt Rd., Taipei 10617, Taiwan; emails: nielsreijers@gmail.com, cshih@csie.ntu.edu.tw.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Association for Computing Machinery.
1550-4859/2019/08-ART34 $15.00
https://doi.org/10.1145/3341170

ACM Transactions on Sensor Networks, Vol. 15, No. 3, Article 34. Publication date: August 2019.
less powerful, but also much cheaper and low power enough to potentially last for months or years on a single battery. For the first-class normal operating systems, languages and compilers can be used, but in this article, we focus specifically on the latter class for which no such clear standards exist. Our experiments were performed on an ATmega128: a 16MHz 8-bit processor, with 4kB of RAM and 128kB of Flash programme memory, but the approach should yield similar results on other CPUs in this category.

There are two main advantages to using a VM on a sensor node. First, it can offer a safe execution environment, preventing buggy or malicious code from disabling the device. A second advantage is platform independence. While early wireless sensor network applications often consisted of homogeneous nodes, current Internet-of-Things/Machine-to-Machine applications are expected to run on a range of different platforms. A VM can significantly ease the deployment of these applications.

While current VMs offer an impressive set of features, almost all sacrifice performance. The VMs for which we have found concrete performance data 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. Perhaps more importantly, one of the main reasons for using such tiny devices is their extremely low power consumption. For many applications the CPU will 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 CPU has to stay active 10× to 100× longer because of the slowdown incurred by using a VM, this may suddenly become the dominant factor.

Mercury. As an example, one of the few applications reporting a detailed breakdown of its power consumption is Mercury [32], a platform for motion analysis. The greatest energy consumer is the sampling of a gyroscope, at 53.163mJ. Only 1.664mJ 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 significant or even by far the largest energy consumer. A more complex operation such as a 512-point FFT costs 12.920mJ. For tasks like this, even a slowdown by a much smaller factor will have a significant impact on the total energy consumption.

Lossless Compression. LEC [33] is a lightweight lossless compression algorithm designed specifically for sensor nodes that aims to reduce the cost of transmitting sensor data. Using the datasheets for the ATmega128 [34] CPU and Chipcon CC2420 radio [7], we estimate the energy per CPU cycle in active mode to be 2.53nJ/cycle, and the energy per bit transmitted at 288.2nJ/bit, assuming an effective bitrate of 165kbps [26].

We used LEC to compress 256 16-bit ECG measurements [38]. This saved 2,256 bits, or 651μJ, on transmitting the data, at the expense of 97,052 CPU cycles, or 246μJ. A VM-induced slowdown of more than 651/246 = 2.6 means more energy will be spent on compression than saved on transmission. The exact ratio depends on many parameters, but in many scenarios a slowdown of 10–100× will make even the lightweight LEC compression a net loss.

Amulet. A third example is the smart watch platform Amulet [18]. Amulet uses the MSP430 CPU to provide a month-long battery lifetime. The authors envision a future with multiple vendors of Amulet devices, using different hardware platforms, and developers submitting applications to an app store. In such a scenario, the platform independence provided by a VM is a valuable property.

The authors report that “energy consumption is significantly impacted by the fraction of time the application microcontroller is active” and that, depending on the application, the CPU is active and executing application code for 0.8% to 11.8% of the time. Thus, the 10–100× slowdown of a
VM will increase total energy consumption, and, depending on the selected applications and exact slowdown, in many cases the CPU may not have enough time to complete its tasks.

AOT 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 runtime, 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 runtime, which 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 to AVR compiler 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 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, we focus on the middle option: translating the bytecode to native code on the device itself at loadtime. The main research questions to answer are: How close an AOT compiling sensor node VM can come to native C performance, what optimisations are necessary to achieve this, and what tradeoffs are involved?

2 RELATED WORK

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 [28] that provide specialised instructions for a specific problem domain. One of the first VMs proposed for sensor networks, Maté [27], is an ASVM. It provides single instructions for tasks that are common on a sensor node, so programmes can be very short. Unfortunately, they have to be written in a low-level assembly-like language, limiting its target audience. SwissQM [35] is a more traditional VM, based on a subset of the Java VM but extended with instructions to access sensors and do data aggregation. VM* [23] 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 have also been developed, allowing the programmer to use general-purpose languages like Java, Python, or even LISP [4, 6, 17]. The smallest official Java standard is the Connected Device Limited Configuration [37], 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.

Only a few papers describing sensor-node VMs contain detailed performance measurements, summarised in Table 1. For SensorScheme, we omitted a benchmark that only calls a natively implemented random number generator at 4× slowdown, since this only tells us a function call takes 3× longer than generating the random number. TinyVM is a special case, since it is a VM for C that implements part of an application in compact bytecode and performance-critical parts in native machine code, thus trading platform independence for performance. An earlier version of the article, which compares fully native and fully interpreted versions of their benchmarks, reports...
Table 1. Slowdown for Interpreting Sensor-Node VMs

| VM          | Source                  | Platform     | Performance vs. native C |
|-------------|-------------------------|--------------|--------------------------|
| Darjeeling  | Delft University of Technology | ATmega128    | 30×–113× slower [6]     |
| TakaTuka    | University of Freiburg Mica2 (AVR) and JCreate (MSP) | 230× slower [10] |
| TinyVM      | Yonsei University ATmega128 | 2.5×–16× slower [19] |
| DVM         | UCLA ATmega128L         | 108× slower [5] |
| SensorScheme| University of Twente MSP430 | 555× slower [25] |

a slowdown between 14× and 72× [20]. 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 [36]. 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 [9]. Although the initialised objects are translated to C structures that are compiled and linked into a single image, the bytecode is still interpreted. While in general we can produce higher-quality code when compiling offline, doing so sacrifices the key advantages of using a VM.

Hsieh et al. describe an early ahead-of-time compiling desktop Java VM [21], 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 runtime, a slower optimising compiler may be used to speed up critical methods.

Since JIT compilers work at runtime, much effort has gone into making the compilation process as lightweight as possible, for example [24]. More recently, these efforts have included JIT compilers targeted specifically at embedded devices. Swift [45] is a lightweight 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 [14] 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 the literature require too much memory. Indeed, some authors suggest sensor nodes are too restricted to make AOT or JIT compilation feasible [3, 44].

VM performance has been studied extensively on the desktop and for more powerful embedded systems, but for sensor-node VMs this aspect has been mostly ignored. To the best of our knowledge, AOT compilation on a sensor node has only been done by Ellul and Martinez [11], and our work builds on their approach. They improve performance considerably compared to the interpreters, but there is still much room for improvement. Using our benchmarks, their approach
generates code that is up to 9.1× slower and 4.5× larger than optimised native C. There are three reasons why it is important to improve on these results: First, the loss of performance results in an equivalent increase in CPU power consumption, thus reducing battery life. In the case of LEC compression mentioned in the introduction, a slowdown of over 2.6× would result in a net energy loss. Second, the reduced performance results in a reduction in the maximum sampling rate for sensing applications, and for the Amulet smart watch a 9.1× slowdown in the application code means the CPU is not fast enough to finish some of the application’s tasks in time [18]. Finally, the increased size of the compiled code reduces the amount of code we can load onto a node. Given that Flash memory is already restricted, this is a major sacrifice to make when adopting AOT on sensor nodes.

This article makes the following contributions:

- We identify the major sources of overhead when using the baseline approach as described by Ellul and Martinez.
- 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 reduce the code size overhead by 56%, which quickly compensates for the increase in VM size, thus mitigating a drawback of the previous AOT approach.
- The optimisations also eliminate most of the performance overhead caused by the JVM’s stack-based architecture, and over 80% of performance overhead overall.
- 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 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 standard CoreMark benchmark [8] and two real-world sensor-node applications.

3 AHEAD-OF-TIME TRANSLATION

Our implementation is based on Darjeeling [6], a Java VM for sensor nodes, running on an Atmel ATmega CPU. Like other sensor-node VMs, it is originally an interpreter. We add an AOT compiler to Darjeeling: 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 [10], it depends on the ability to execute code from RAM, which many embedded CPUs, including the ATmega, cannot do.

The process from Java source to a native application on the node is shown in Figure 1. Like all sensor-node JVMs, Darjeeling uses a modified JVM bytecode. Java source code is first compiled to normal Java classes, which are optimised by ProGuard [40]. The optimised Java classes are then
transformed into Darjeeling’s own format, called an “infusion.” For details of this transformation, we refer to the Darjeeling paper [6]. Here it is sufficient to know that the bytecode is modified to make it more suitable for execution on a tiny device, for example, by adding 16-bit versions of most operations, but the result remains very similar to standard JVM bytecode. It is also important 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 it is translated to native AVR code at load time.

We made several modifications to Darjeeling’s 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 BRTARGET opcode, used to mark the target locations of branch instructions; all branch instructions target a BRTARGET id instead of a bytecode offset
- the MARKLOOP opcode, used to mark inner loops and list the variables used in the loop
- _FIXED versions of the GETFIELD_A and PUTFIELD_A opcodes, used to access an object’s reference fields when the offset is known at compile time
- the SIMUL opcode, used for 16 × 16-bit to 32-bit multiplication
- modified array access opcodes to use 16-bit indexes
- _CONST versions of the bit shift opcodes to support constant shifts
- the INVOKELIGHT opcode for an optimised “lightweight” way of calling methods

3.1 Goals and Limitations

Working on resource-constrained devices means we have to make some compromises. Our main goal is to build a 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. We would like to be able to support scenarios were multiple applications are running on a single device, so when new code is being loaded, the impact on other applications should be as small as possible.

Therefore, the translation process should be very lightweight. Specifically, it should use as little memory as possible, since memory is a scarce resource. This means we cannot do any analysis on the bytecode that would require us to hold complex data structures in memory. When receiving a large programme, we should not have to keep multiple messages in memory but will 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.

Bytecode instructions are processed in a single pass, one instruction at a time. Only some small, fixed-size data structures are kept in memory during the process. A second pass over the generated code then fills in addresses left blank by branch instructions, since the target addresses of forward branches are not known until the target instruction is generated.

Additional optimisations, beyond those described here, may further improve performance, but on resource-constrained devices where CPU cycles, energy, RAM, and Flash memory are all restricted, there is always a tradeoff. While the 1 to 2 orders of magnitude slowdown seen in interpreters may rule out the use of a VM in many real-world scenarios, the average 1.7× slowdown after our optimised AOT compilation will be acceptable for most applications.

Adding more complex optimisations may rule out the use of a VM in scenarios that cannot afford the increase in VM size or the additional state that needs to be kept in RAM, while the
improvement in performance is unlikely to enable many extra applications and has only limited impact on the total energy consumption. Thus, we argue the set of optimisations presented here and our goal of using as little resources as possible are at a sensible point in this tradeoff.

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 runtime 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 AOT compilation.

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 [10], 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 more lightweight models for error handling and concurrency, similarly to the models used in nesC and Amulet [15, 18], more appropriate on a sensor node.

3.2 Translating Bytecode to Native Code

The basic approach to translate bytecode to native code on a sensor node was first described by Ellul and Martinez [11]. When we receive a bytecode instruction, we simply replace it with an equivalent sequence of native instructions, using the native stack to mimic the JVM stack. An example is shown in Table 1.

The first column shows a fragment of JVM code that does a shift right of variable A and repeats this while A is greater than B. While not a very practical function, it is the smallest example that will allow us to illustrate our code generation optimisations. The second column shows the code the AOT compiler will execute for each JVM instruction. Together, the first and second columns match the case labels and body of a switch statement in our compiler. The third column shows the resulting AVR native code, which is currently almost a one-on-one mapping, with the exception of the branch and some small optimisations by a simple peephole optimiser, both described below.

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

3.2.1 Peephole Optimisation. From Table 2, it is clear that this approach results in many unnecessary push and pop instructions. Since the JVM is a stack-based VM, each instruction must obtain its operands from the stack and push any result back onto it. As a result, almost half the instructions are push or pop instructions.

To reduce this overhead, Ellul proposes a simple peephole optimiser [10]. The compilation process results in many push instructions that are immediately followed by a pop. If they target the same register, then they have no effect and are removed. If the source and destination registers differ, then the two instructions are replaced by a move. The result is shown in the third column of Table 2. Two push/pop pairs have been removed, and one has been replaced by a move.

3.2.2 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 one to three words, depending on the distance to the target, so it is also not known how much space should be reserved for a branch.
Table 2. Translation of \( \text{do}\{A\gg\gg=1;\} \text{while}(A>B); \)

| JVM          | AOT compiler | AVR  | cycles |
|--------------|--------------|------|--------|
| 0: BRTARGET(0) « record current address » | emit_LDD(R1,Y+0) | LDD R1,Y+0 | 4 |
|              | emit_PUSH(R1) | PUSH R1   | 4 |
| 1: SLOAD_0   | emit_LDI(R1,1) | LDI R1,1 | 2 |
|              | emit_PUSH(R1) | MOV R2,R1 | 1 |
| 2: SCONST_1  | emit_PUSH(R2) | POP R1 | 4 |
|              | emit_RJMP(+2) | RJMP +2 | 2 |
| 3: SUSHR     | emit_LSR(R1) | LSR R1 | 2 |
|              | emit_DEC(R2) | DEC R2 | 2 |
|              | emit_BRPL(-2) | BRPL -2 | 3 |
| 4: SSTORE_0  | emit_PUSH(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_PUSH(R1) | POP R2 | 4 |
|              | emit_CP(R1,R2) | CP R1,R2 | 2 |
|              | emit_branchtag(GT,0) | BRGT 0: (taken), 2 |
|              | | or 1 (not taken) |

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 a branch target id instead of a bytecode offset. When the VM encounters a BRTARGET during compilation, no code is emitted, but the address where the next instruction will be emitted is recorded in a separate part of Flash memory. Branch instruction initially emit a temporary three-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 to replace 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 three words, but this causes a performance penalty on small, non-taken branches. Instead, the VM does another scan of the code before replacing the branch tags and updates the branch target addresses to compensate 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 they often also result in smaller generated code.

### 3.3 Darjeeling Split-stack Architecture

In the JVM, all objects are stored in the heap and are freed by the garbage collector when they are no longer used. To determine which objects can be freed, the garbage collector needs to determine
which objects are still used by the programme. It does so by first determining a root set containing all objects that are directly referenced from a local or static variable, or from the operand stack. This set is then incrementally extended to include objects referenced from the current set, until no more objects can be added. Any remaining objects can then be garbage collected.

To determine the root set, the VM must know which fields (variables or operand stack slots) are references and which are normal integers. To do so efficiently, Darjeeling separates reference and integer values throughout the VM, as shown in Figure 2. In a normal JVM the same slot in the operand stack could at different times contain an integer or a reference. Therefore Darjeeling splits the operand stack into two separate stacks for integers and references.

In our AOT compiler we use the native stack for the JVM integer operand stack, while space for the reference stack is reserved in the stack frame. 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 a stack pointer for the reference stack.

3.4 Target Platforms

The AVR family of CPUs is widely used in low-power embedded systems. We implemented our VM for the ATmega128 CPU. 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 eight-bit registers. We ran several experiments where we restrict the number of registers our VM may use. As is often the case with caches, we found the first few available registers to have the largest impact, while the added improvement is reduced for each added register. Based on this, 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 [11], to both be good matches as well.

4 SOURCES OF OVERHEAD

The performance of this basic approach is still far behind optimised native C. To improve performance, it is important to identify the causes of this overhead. The main sources of overhead we found are as follows:

- 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 them.

4.1 Lack of Optimisation in javac

A first source of overhead comes from the fact that the standard javac compiler does almost no optimisations. 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. Since the JVM is an abstract machine, there is no clear performance model to optimise for. Runtime 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.

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 runtime 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 lack of optimisations in javac leads to significant overhead.

We do use the ProGuard optimiser [40], 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.

4.2 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.

4.2.1 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 2, 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 corresponding push and pop instructions will not be consecutive, and the peephole optimiser cannot eliminate these cases.

4.2.2 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 soon needed again. 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 2, it is clear that the SLOAD_0 instruction at label 5 is unnecessary, since the value is already in R1.

4.2.3 Type 3: JVM Instruction Set Limitations. A final source of overhead 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 as in native code. Given enough processing power, compilers can do the complex transformations necessary to make compiled JVM code run almost as fast as native C, but a sensor node does not have such resources and must simply execute the instructions as they are.
In Table 2, we see that there is no way to express a single bit shift directly. Instead, a constant value is loaded onto the stack and executed, followed by a 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 that for each iteration the VM has to load the reference and index back onto the stack and redo the address calculation. In contrast, the native C version would typically just slide a pointer over the array.

4.3 Method Call Overhead

The final source of overhead comes from method calls. In the JVM, each method has a stack frame (or “activation frame”) that 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).” [16]

Darjeeling’s stack frame layout is shown in Figure 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 6.2 and 6.3)
(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 540 cycles for the simplest case: a call to a static method without any parameters or return value. For a virtual method the cost is higher, because we need to look up the right implementation. While we may be able to save some more cycles with an even more rigorous refactoring, it is clear that the number of steps involved will always take considerably more time than a native function call.

4.4 Optimisations

Having identified these sources of overhead, the next three sections will describe a set of optimisations to address them. Table 3 lists each optimisation and the source of overhead it aims to reduce. The following sections will discuss each optimisation in detail.
Table 3. List of Optimisations per Overhead Source

| Source of overhead | Optimisation |
|--------------------|--------------|
| Section 5          | Lack of optimisations in javac Manual optimisation of Java source code |
| Section 6          | 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 shift optimisation |
|                    |                            | 16-bit array indexes       |
| Section 7          | Method call overhead        | INVOKELIGHT instruction    |

5 MANUALLY OPTIMISING THE JAVA SOURCE CODE

As shown in Section 3, 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” that uses normal, well-known optimisation techniques to produce better-quality bytecode.

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 important to mention them explicitly and analyse their impact, since many developers may expect many of these to happen automatically, and without them it would be impossible to reproduce our results.

The most common optimisations we performed are as follows:

- manually inlining small methods
- prefer to use 16-bit variables for array indexes where possible
- 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 loops optimisation discussed in Section 6.4, prefer to store a value in a local variable if it may be used again soon rather than accessing the array or object twice
- use bit shifts for multiplications by a power of two

We will examine the effect of these optimisations on the Core Mark benchmark in Section 8.1.

Manual Inlining. We manually inline all small methods that were either a #define in the original C code or a function that was inlined by avr-gcc. ProGuard can also inline small methods, but when it does, it simply replaces the INVOKELIGHT instruction with the callee’s body and adds 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 automatically.
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, then 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 based on very conservative assumptions that would work well for most devices in this class.

Example. Listing 1 shows an example of these manual optimisations, applied to the bubble sort benchmark. To have a fair comparison, we applied exactly the same optimisations to the C versions of our benchmarks, but this had little or no effect on the performance.

6 OPTIMISATIONS: REDUCING 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.

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

PUSH Rs

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

POP Rd

In this case, the pair can be eliminated if Rs == Rd; otherwise, it is replaced by a “mov Rd, Rs.” Two push/pop pairs remain in Table 2. The pair in instructions 5 and 7 pops to 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.

6.2 Simple Stack Caching
The improved peephole optimiser can remove part of the type 1 overhead, but still many cases remain where it cannot eliminate the push/pop instructions. We use a form of stack caching [12] to eliminate most of the remaining push/pop overhead. Stack caching is not a new technique, but
the tradeoffs 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, otherwise the overhead from managing it at runtime will outweigh the time saved by reducing memory accesses. Since we only use the cache state at load time, this restriction does not apply for an AOT compiler, and we can afford to spend more time managing the cache. 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, then 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 original 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 need a free register to load the value into, which will later be pushed onto the stack. If all registers are in use, then getfree spills the register that is lowest on the stack to memory by emitting a PUSH and then returns that register. This means the top of the stack is kept in registers, while lower elements are spilled to memory when necessary.
- **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, then 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 4, we translate the same example we used before but this time using stack caching. To save space, Table 4 also includes the constant shift optimisation described in Section 6.5.4. 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,” and so on (this example does not use the reference stack). In the next two optimisations, we will extend the cache state further.

The example only shows three registers, but the ATmega128 has 32 8-bit registers. Since Darjeeling uses a 16-bit stack, they are managed as pairs. Ten 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. The cache state is known if it was reached from the previous instruction but not if it was reached through a branch. To ensure the cache state is the same on both paths, the stack cache is flushed to memory whenever we encounter either a branch or a BRTARGET instruction.
Table 4. Simple Stack Caching

| JVM                      | AOT compiler | AVR cycles | cache state R1 | cache state R2 | cache state R3 |
|--------------------------|--------------|------------|----------------|----------------|----------------|
| 6: BRTARGET(0)           | « record current address » |            |                |                |                |
| 1: SLOAD_0               | operand_1 = sc_getfreereg() | emit_LDD(operand_1,Y+0) | LDD R1,Y+0 4   | Int1           |                |
|                          |               | sc_push(operand_1) |                |                |                |
| 3: SUSHR CONST(1)        | operand_1 = sc_pop() | emit_LSR(operand_1) | LSR R1 2       | Int1           |                |
|                          |               | sc_push(operand_1) |                |                |                |
| 4: SSTORE_0              | operand_1 = sc_pop() | emit_STD(Y+0,operand_1) | STD Y+0,R1 4   | Int1           |                |
| 5: SLOAD_0               | operand_1 = sc_getfreereg() | emit_LDD(operand_1,Y+0) | LDD R1,Y+0 4   | Int1           |                |
|                          |               | sc_push(operand_1) |                |                |                |
| 6: SLOAD_1               | operand_1 = sc_getfreereg() | emit_LDD(operand_1,Y+2) | LDD R2,Y+2 4   | Int2           | Int1           |
| 7: IF SCMPGT 0:          | operand_1 = sc_pop() | emit_CP(operand_1, operand_2) | CP R2,R1 2     |                |                |
|                          | operand_2 = sc_pop() | emit_branchtag(GT, 0); | BRGT 0: 2 or 1  |                |                |

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.

6.3 Popped Value Caching

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

A "value tag" is added to each register’s cache state to keep track of what value is currently held in the register, 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), and so on. These tags are encoded in a 16-bit value.

A function sc_can_skip is added to the cache manager. This function will examine the type of each instruction, its value tag, and the cache state. If it finds that a value that needs to be loaded is already present in a register, then 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 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 popdestructive. 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
Table 5. Popped Value Caching

| JVM | AOT compiler | AVR cycles | cache state R1 | cache state R2 | cache state R3 |
|-----|--------------|-------------|----------------|----------------|----------------|
| 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 LS0       |
| 3:  | SUSHR_CONST(1) | `operand_1 = sc_pop_destructive()` | `emit_LSR(operand_1)` | LSR R1 2 |               |
|     |              |             | `sc_push(operand_1)` |               | Int1           |
| 4:  | SSTORE_0     | `operand_1 = sc_pop_tostore()` | `emit_STD(Y+0,operand_1)` | STD Y+0,R1 4 | LSO            |
|     |              |             | `sc_push(operand_1)` |               | Int1 LS0       |
| 5:  | SLOAD_0      | "skip codegen, just update cache state" |             |               |
| 6:  | SLOAD_1      | `operand_1 = sc_getfreereg()` | `emit_LDD(operand_1,Y+2)` | LDD R2,Y+2 4 | Int1 LS0       |
|     |              |             | `sc_push(operand_1)` |               | Int2 LS0 Int1 LS1 |
| 7:  | IF_SCMPTGT 0: | `operand_1 = sc_pop_nondestructive()` | `emit_CP(operand_1,operand_2)` | CP R2,R1 2 | LSO            |
|     |              |             | `operand_2 = sc_pop_nondestructive()` |               | LSO            |
|     |              |             | `emit_branchtag(GT,0)` | BRGT 0: 2 or 1 | LSO            |

containing LS0, then the cache manager would clear that tag, since it is no longer accurate after the variable is updated.

In line 5, LS0 is loaded again, but now the cache state shows that LS0 is already in R1. This means the value does not need to be loaded from memory, but the cache state can simply be updated so that R1 is pushed onto the stack. At runtime, this SLOAD_0 will have no cost at all.

There are a few more details to get right. For example, if a value is loaded that is already on the stack, `sc_can_skip` generates 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, then the least recently used register is returned. This is done to maximise the chance a value can be reused later, since recently used values are more likely to be used again.

Branches. Since the state of the registers is not known if an instruction is reached through a branch, all value tags are cleared when passing a BRTARGET instruction, meaning that any new loads will have to come from memory. At branches the value tags are kept, because if the branch is not taken, the state of the registers is known in the next instruction.

6.4 Mark Loops

Popped value caching reduces the type 2 overhead significantly, but the fact that the value tags are cleared at branch targets means that a significant 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 addressed by the next optimisation.

Again, the infuser is modified 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 it encounters 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, then code is generated to prefetch these variables from memory and store them in registers. While in the loop, loading
Table 6. Mark Loops

| JVM | AOT compiler | AVR | cycles | cache state R1 | cache state R2 | cache state R3 |
|-----|--------------|-----|--------|----------------|----------------|----------------|
| 0: MARKLOOP(0,1) | « emit markloop prologue: L0 and L1 are live » | LDD R1,Y+0 4 | LS0 PIN | | | |
| 1: BRTARGET(0) | « record current address » | LDD R2,Y+2 4 | LS0 PIN | LS1 PIN | | |
| 2: SLOAD_0 | « skip codegen, just update cache state » | | | | | |
| 3: SSTORE_0 | « skip codegen, move to pinned reg » | MOV R1,R3 1 | LS0 PIN | LS1 PIN | | |
| 4: SLOAD_1 | « skip codegen, just update cache state » | | | | | |
| 5: IF_SCMPGT 0: | operand_1 = sc_pop_non destructive() | MOV R3,R1 1 | LS0 PIN | LS1 PIN | | |
| 6: IF_SCMPGT 0: | operand_2 = sc_pop_nondestructive() | MOV R3,R1 1 | LS0 PIN | LS1 PIN | | |
| 7: MARKLOOP(end) | « emit markloop epilogue: L0 is live » | STD Y+0,R1 4 | LS0 PIN | LS1 PIN | | |

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 means more load instructions can be eliminated. In addition, store instructions are replaced by a much cheaper move to the pinned register. INC instructions, which increment a local variable, operate directly on the pinned register, saving both a load and a store. All these cases are handled in sc_can_skip, bypassing the normal code generation. A small change is needed to the sc_pop_destructive function. If the register that is about to be popped is pinned, then it cannot just return this register, since that would corrupt the value of the pinned local variable. Instead, first a move to a free, non-pinned register is emitted, and this register is returned instead.

In Table 6 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, SUSRH_CONST pops destructively. Simply returning register 1 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, SSTORE_0 can be skipped, but we do need to emit a move back to the pinned register.

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

Finally, the loop ends with another MARKLOOP, telling the compiler only local 0 is live at this point. This means LS0 in register R1 needs to be stored back to memory, but LS1 can be skipped, since it is no longer needed.

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 2.

6.5 Instruction Set Modifications

Next, we introduce four optimisations that target the type 3 overhead: cases where limitations in the JVM instruction set means some operations cannot be expressed as efficiently as in native code. This type of overhead is the most difficult to address, because many of the transformations a desktop VM can do to avoid it require more resources than a tiny device can afford. Additionally, this type of overhead covers many different cases, and optimisations that help in a specific case may not be general enough to justify spending extra resources on it.

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

Darjeeling’s original instruction set already differs 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-bitslots 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 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 other modifications to Darjeeling’s instruction set:

6.5.1 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 3, integer fields will be at the same offset, regardless of whether a runtime 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 at runtime, adding significant overhead.

This overhead can be avoided if the offset is known 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 field’s offset at AOT compile time. Conveniently, one of the optimisations ProGuard does is to mark any class that is not subclassed as final, so most of this is automatic.

Alternative Solutions. An alternative that was 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.

The current solution was chosen 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, almost all classes in our benchmark could be marked final, so the _FIXED opcodes could be used in most cases. 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, and not a direct result of the AOT techniques or the JVM’s design. Therefore, all results reported in this contribution are with this optimisation already turned on.

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

6.5.2 **SIMUL 16-bit x 16-bit to 32-bit Multiplication.** While Darjeeling already introduced 16-bit arithmetic operations, it does not cover the case of multiplying two 16-bit shorts and storing the result in a 32-bit integer. In this case, the infuser would emit S2I instructions to convert the operands to two 32-bit integers and then use the normal IMUL instruction for full 32-bit multiplication. On a device with a shorter word size, this is significantly more expensive than 16 x 16 to 32-bit multiplication.

A new opcode, SIMUL, was added 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.

More instructions could be added, for example, SIADD instruction for addition, BSMUL for 8-bit to 16-bit multiplication, and so on. But there is always a tradeoff between the added complexity of an optimisation and the performance improvement it yields, and for these cases this is much smaller than for SIMUL.

6.5.3 **16-bit Array Indexes.** The 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, such large indexes are never needed, so the array access instructions were modified to take 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. 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.

6.5.4 **Constant Bit Shifts.** Finally, shifts by a constant number of bits appear in 9 of the 12 benchmarks described in Section 8. They appear not only in computationally 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 shifts by a constant number of bits: The constant needs to be pushed onto the stack, and the shift is implemented as a loop that shifts one bit at a time. If the number of bits to shift by is known at compile time, then much more efficient code can be generated.
Note that this is different from other arithmetic operations with a constant operand. For operations such as addition, our translation process results in loading the constant and performing the operation, similar to what avr-gcc produces in most cases. An addition takes just as long when both operands are variables, as it does when one 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 that can be generated for a shift by a constant number of bits.

These cases are optimised by adding _CONST versions of the bit shift instructions ISHL, ISHR, IUSHR, SSHL, SSHR, and SUSHR. The infuser does a simple scan of the bytecode 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 1-byte operand 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 three of the benchmarks used in Section 8.

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

As mentioned before, this optimisation was already included in the example from Table 4 onward, so the effect can be seen by comparing the SCONST_1 and SUSHR instructions in Table 2 and the SUSHR_CONST instruction in Table 4.

7 OPTIMISATIONS: METHOD CALLS

Finally, we will look at the overhead caused by method calls. In native code, the smallest functions only need 8 cycles for a CALL and RET, and some MOVs 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 seen in Section 4.3, a considerable amount of state needs to be initialised for a Java method call. For the simplest methods this takes about 540 cycles, which increases further for large methods with many parameters.

The methods in a programme typically form a spectrum from large methods at the base of the call tree that take a long time to complete and are called only a few times, to small (near-) leaf methods that are fast and frequently called. Figure 4 shows this spectrum for the CoreMark benchmark.

For the slow methods at the base, the overhead of the method call is not very significant compared to the overall execution time, and we can afford to take the 540 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, inlining larger methods that are called from multiple places can lead to a significant increase in code size. 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.
7.1 Lightweight Methods

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

- no stack frame is created for lightweight methods, but the caller’s frame is used
- parameters are passed on the stack, rather than in local variables

Lightweight methods offer a third choice, in between a normal method call and method inlining. When calling a lightweight method, the method’s code is CALLED directly, bypassing the VM completely. The caller’s stack frame is reused, and the parameters are left on the stack, instead of being put in local variables. In effect, the lightweight method behaves similarly to inlined code but does not incur the code size overhead of inlining large methods.

Because the method will be called from multiple locations, which have different cache states, the stack cache has to be flushed to memory before a call. This results in slightly more overhead than for inlined code but much less than for a normal method call.

As an example, consider the simple isOdd method in Listing 2:

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

Listing 2. Simple, stack-only lightweight method example.

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

A new instruction, INVOKELIGHT, is introduced to call lightweight methods. In the bottom half of Listing 3, 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.

7.1.1 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. For a
lightweight method that uses local variables, the caller needs to reserve space for them in its stack frame, equal to the maximum number of slots needed by all the lightweight methods it may call.

In the AOT compiled code, the ATmega’s Y register points to 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.

7.1.2 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 32-bit crcu32 function, which is implemented as two calls to crcu16. While crcu16 is the most critical, there is still one call to crcu32 for every two to crcu16.

So far we have not discussed how to handle the return address in a lightweight method. The AOT compiler uses the native stack to store JVM integer stack value, which means the integer operands to a lightweight method will be on the native stack. But after a native CALL instruction, 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 registers 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, it is saved in the first local variable slot, and Y is incremented to skip this slot. Since each lightweight method has its own block of locals, calls can be nested as deep as necessary. 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.

7.1.3 Stack Frame Layout. A normal method that invokes a possible sequence 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, 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

Listing 3. Comparison of lightweight and normal method invocation.

```
// NORMAL INVOCATION
// INVOKESTATIC isOdd:
push r25  // Flush the cache
push r24
call &preinvoke // Save X and SP
ldi r22, 255  // Set parameters
ldi r23, 2  // for callMethod
ldi r24, 21
ldi r20, 64
ldi r21, 42
ldi r18, 13
ldi r19, 9
ldi r25, 2
call &callMethod // Call to VM
call &postinvoke // Restore X and SP
```

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

7.1.4 Mark Loops. 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 those variables are saved back to memory before calling the lightweight method and reloaded 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 loops are allocated low to high, and for normal stack caching from high to low, in many cases the two may not collide.

7.1.5 Example Call. An example of the most complex case for a lightweight call is shown in Listing 4, which shows how method f from Figure 5 would call g_lw, assuming f is in a markloop block at the time that pinned a variable to R14:R15, and these registers are also used by g_lw.

In the translation of the INVOKELIGHT instruction, first the stack cache is flushed to memory, and then the value of the local variable at offset 22 is saved because it was pinned to R14:R15. Next, the Y register is incremented to skip the caller’s own local variables and point to the start of the space reserved for lightweight method locals.

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

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

Fig. 5. Stack frame layout for a normal method f, which calls lightweight method g_lw, which in turn calls lightweight method h_lw.
7.2 Creating Lightweight Methods

Two ways to create a lightweight method are supported:

- handwritten JVM bytecode
- converting a Java method

7.2.1 Handwritten JVM Bytecode. For the first option methods are declared native in the Java source code, so the code calling it will compile as usual. The infuser is provided with a handwritten implementation in JVM bytecode, which it will simply add to the infusion and then process it in the same way it processes a normal method, with one added step.

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.

7.2.2 Converting Java Methods. This handwritten approach is useful for the smallest methods and can be used 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, normal Java methods can be converted to lightweight methods by adding a @Lightweight annotation to it.

The infuser will scan the methods in an infusion for this annotation. When it finds a method marked @Lightweight, the transformation to turn it into a lightweight method is simple: In front of the original method body, the infused adds a block of dummy LW_PARAMETER instructions, followed by STORE instructions to pop the method’s parameters off the stack and store them in the right local variables. After this, the method can be called as a lightweight method.

Listing 5 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 runtime cost. Another disadvantage of the converted method is that it has to use a local variable, which will slightly increase memory usage, but in return this approach offers a very easy way to create lightweight methods.

7.2.3 Replacing INVOKEs. 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

Listing 4. Full lightweight method call.
7.3 Overhead Comparison

The overhead for the various ways to call a method are compared in Table 7. Manually inlining code yields the best performance but at the cost of increasing code size if larger methods are inlined. ProGuard inlining is slightly more expensive, because it always saves parameters in local variables.

Both lightweight methods options cause some overhead, although this is very little compared to a full method call. First, we need to flush the stack cache to memory to make sure the parameters are on the real stack. This takes two push and eventually two corresponding pop instructions per word, costing eight 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 seven pairs to pin variables in a MARKLOOP block.

After calling the method, the method prologue for a lightweight method is simple. It saves the return address that is restored in the epilogue. This takes 8 cycles if the address can be left in registers or 16 if it needs to be stored in a local variable slot.

For small handwritten lightweight methods, this is the only cost, but for larger ones created by converting a Java method, STORE instructions are added to copy the parameters from the stack into local variables, as shown in Listing 5. This is similar to the overhead incurred by ProGuard’s method inlining and costs four cycles per word for the STORE and possibly four more if the corresponding LOAD cannot be eliminated by popped value caching.

Listing 5. Comparison of handwritten lightweight method and converted Java 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           | -80                    |                           |                   |
| create stack frame|                 |                   | -450                   |                           |                   |
| method pro-/epilogue | 8 or 16       | 8 or 16           | 10 to 71               |                           |                   |
| store and load parameters | 4 or 8 per word | 4 or 8 per word | 4 or 8 per word        |                           |                   |
| total            | 4 or 8 per word | 16 to 84 +        | 16 to 84 +             | -540 to -601 +           |                   |

\(^a\)Excluding the effect on future popped value cache performance because of cleared value tags.
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 runtime, limiting the relative overhead.

The number of cycles in Table 7 is just a broad indication of the overhead. Some factors, such as the cost of clearing the value tags is hard to predict, and inlining may allow some optimisations that are not possible with a method call. In practice, the actual cost varies but is in the predicted range.

The cost of normal method call 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 2) are fixed, meaning a call will cost at least around 540 cycles, which increases to over 700 cycles for more complex methods taking many parameters.

### 7.4 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 are in a lightweight call, then 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.** Lightweight virtual methods are not supported, but this is something that could be considered in future work. However, in all benchmarks, virtual methods were only used in a single case that could easily be transformed into code using static methods (see Section 8.1.2).

**Stack Frame Usage.** Finally, while many methods can be made lightweight, a method calling a lightweight method will always reserve space for the lightweight method’s local variables in its stackframe. This space is reserved, regardless of whether the lightweight method is currently executing or not, and the more nested lightweight calls are made, the more space needs to be reserved.

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

### 8 EVALUATION

A set of 12 different benchmarks is used to measure the effect of the optimisations:

- **bubble sort:** taken from the Darjeeling sources, and used in References [6, 10]
- **heap sort:** standard heap sort [2]
- **binary search:** taken from the TakaTuka [4] source code
• \textit{xxtea}: as published in Reference [43]
• \textit{md5}: also taken from the Darjeeling sources, and used in References [6, 10]
• \textit{rc5}: taken from LibTomCrypt [29]
• \textit{CoreMark 1.0}: a freely available benchmark developed by EEMBC [8]
• \textit{FFT}: taken from the Harbor source code [42]
• \textit{Outlier detection}: implemented as described in Reference [25]
• \textit{LEC compression}: implemented as described in Reference [33]
• \textit{MoteTrack}: indoor localisation using RSSI [30, 31]
• \textit{Heat detection}: an application used in our group to track persons and detect fires using an $8 \times 8$ pixel heat sensor, split into a calibration and detection phase.

The first six 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. The processing they do is different for each benchmark, allowing us to examine how our optimisations respond to different kinds of code. The sort, search, and FFT benchmarks all operate on 16-bit data.

\textit{CoreMark} is an industry standard synthetic benchmark that demonstrates the VM’s ability to handle large applications. Due to its mix of different kinds of processing, it is a good indication of the average performance.

The next five benchmarks are code that is typical for sensor nodes. \textit{FFT} and \textit{Outlier detection} are common signal processing operations. \textit{LEC compression} is a lightweight compression algorithm developed specifically for sensor nodes. Finally, \textit{MoteTrack} and \textit{Heat detection} are two complete real-world sensor-node applications.

For each benchmark, a Java version was implemented, keeping the Java version as close to the C original as possible. The code was manually optimised as described in Section 5. These optimisations did not affect the performance of the C version significantly, indicating \textit{avr-gcc} already does similar transformations on the original code. javac version 1.8.0, ProGuard 5.2.1, and \textit{avr-gcc} version 4.9.1 were used. The C benchmarks were compiled at optimisation level -O3 the rest of the VM at -Os.

Manual examination of the compiled code produced by \textit{avr-gcc} revealed some minor points where more efficient code could have been generated, but except for the constant shifts mentioned in Section 6.5.4, this did not affect performance by more than a few percent. This leads us to believe \textit{avr-gcc} is a fair benchmark to compare to.

All experiments were done using the cycle-accurate Avrora simulator [41], emulating an ATmega128 processor. Avrora was modified to get detailed traces of the compilation process and of the runtime performance of both C and AOT compiled code.

The main measurement used for both code size and performance is the overhead compared to optimised native C. To compare different benchmarks, this overhead was normalised to a percentage of the number of bytes or CPU cycles used by the native implementation: A 100% overhead means the AOT compiled version takes twice as long to run or twice as many bytes to store.

### 8.1 CoreMark

\textit{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, the Java version was kept as close to the original C code as possible. Compared to the first simple benchmarks, 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, it was wrapped in an object in the Java version. 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 exist, cmp_idx and cmp_complex. Here, initially the most canonical way to do this in Java was chosen, 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 6. 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, the more natural Java approach was chosen by defining two classes to match the structs in C, shown in Listing 6, and creating instances of these to initialise the list. This is also the faster option, because accessing fields is faster than array access. The tradeoff is memory consumption, since each object has its own heap header.

```java
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;

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

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

Listing 6. C and Java version of the CoreMark list data structures.

8.1.1 Manual Optimisations. After translating the C code to Java, we do a number of manual optimisations that 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 8 shows the slowdown over the native C version, broken down into CoreMark’s three main components. The baseline version, using the original Java code and without any optimisation to the AOT compiler, is 810% slower than native C. Even after applying all our other optimisations, the best that can be achieved with the original code is a 327% slowdown, proving the importance of a better optimising infuser.

Next the manual optimisations, as described in Section 5, are applied to the Java source code, as well as a small extra optimisation to crcu8 that 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, and so on. Combined, these optimisations reduce the overhead for the whole benchmark from 327% to 97%.

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.
Table 8. Effect of Manual Source Optimisation on the CoreMark Benchmark

|                         | native C | baseline | optimised AOT, original source | manually inline small methods | use short array index variables | avoid recal. expressions in a loop | reduce array and object access | reduce branch cost in crcu8 | optimised AOT and source |
|-------------------------|----------|----------|--------------------------------|-------------------------------|---------------------------------|----------------------------------|-----------------------------|--------------------------|-------------------------|
| list time$^a$ vs. nat. C| 17.9     | 124.3 (+594%) | 55.9 (+212%)                  | −0.8 (−4%)                    | −0.1 (−1%)                      | +0.1 (+1%)                       | −0.1 (−1%)                   | −3.6 (−20%)              | 51.4 (+187%)                     |
| matrix time$^a$ vs. nat. C | 49.2 | 360.5 (+633%) | 231.4 (+370%)                 | −57.3 (−76%)                  | −104.6 (−213%)                  | −7.6 (−15%)                      | −18.0 (−37%)                 | −0.5 (−1%)               | 63.4 (+29%)                        |
| state time$^a$ vs. nat. C | 18.0 | 289.0 (+1506%) | 75.9 (+322%)                  | −17.4 (−97%)                  | 0.0 (0%)                        | 0.0 (0%)                        | −2.4 (−13%)                  | −3.2 (−18%)              | 52.9 (+194%)                     |
| total time$^a$ vs. nat. C | 85.1 | 774.4 (+810%) | 363.4 (+327%)                 | −55.5 (−65%)                  | −104.7 (−123%)                  | −7.5 (−9%)                      | −20.5 (−24%)                 | −7.3 (−9%)               | 167.9 (+97%)                      |

$^a$In millions of cycles.

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

Table 8 shows that in the native version, over half of the time is spent in the matrix part of the benchmark, but for the final Java version all three parts are much closer together. The state machine and linked list processing both suffer from a much larger slowdown than the matrix part, which by itself would be the third fastest of all our benchmarks.

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

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

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

Combined, this improves the performance of CoreMark to only 59% overhead over native C, right in the middle of the spectrum of the other benchmarks.
These results point out some weaknesses of Java when used as an embedded VM. The lack of cheap function pointers, or a way of allocating small temporary objects or arrays in a method’s stack frame, means there will be a significant overhead in situations where the optimisations used here cannot be applied.

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

### 8.2 AOT Code Generation Optimisations

Next we will look at the effect of our different optimisations for the baseline AOT translation approach for all 12 of our benchmarks.

The trace data produced by Avrora provides a detailed view into the runtime performance and the different types of overhead. The number of bytes and cycles spent on each native instruction is tracked for both the native C and our AOT compiled version. These are then grouped into four categories that roughly match the types of AOT translation overhead discussed in Section 4.2:

- **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 stack caching is introduced, 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.

The overhead from each category is defined 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 normalised 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 Table 9, which also lists the time spent in the VM on method calls.

Figure 6 shows how the optimisations combine to reduce performance overhead. Both the total overhead and the overhead for each instruction category are shown for the average of the 12 benchmarks. Figure 7 shows the total overhead for each individual benchmark. Starting with Ellul’s original AOT approach with only the simple peephole optimiser as a baseline, each of the optimisations is added incrementally to improve performance. We do not compare to Darjeeling’s interpreter, since it was never optimised for performance, making the results rather arbitrary, and
Table 9. Performance Data Per Benchmark

| EXECUTED BYTECODE INSTRUCTIONS (% of total executed bytecode instructions before optimisation) | B.sort | H.sort | Bin.Search | XXTEA | MD5 | RCS | FFT | Outlier | LEC | CoreMark | MoteTrack | HeatCalib | HeatDetect | average |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Load/Store | 79.8 | 71.7 | 58.1 | 44.9 | 43.3 | 41.1 | 61.1 | 69.0 | 59.5 | 54.1 | 70.3 | 51.8 | 48.0 | 57.9 |
| Constant load | 0.2 | 8.1 | 11.0 | 12.5 | 19.1 | 17.6 | 6.4 | 0.6 | 7.9 | 10.0 | 5.4 | 10.1 | 16.6 | 9.7 |
| Processing | 8.0 | 7.8 | 14.8 | 32.4 | 28.9 | 36.6 | 18.0 | 13.0 | 12.7 | 14.0 | 5.9 | 17.9 | 10.3 | 16.9 |
| math | 8.0 | 5.5 | 10.3 | 10.1 | 12.5 | 10.7 | 11.6 | 13.0 | 7.1 | 8.2 | 5.9 | 3.7 | 9.4 | 8.9 |
| bit shift | 0.0 | 2.2 | 4.5 | 8.1 | 5.4 | 8.0 | 6.1 | 0.0 | 3.8 | 2.2 | 0.0 | 8.5 | 0.9 | 3.8 |
| bit logic | 0.0 | 0.0 | 0.0 | 14.2 | 11.0 | 17.9 | 0.3 | 0.0 | 1.9 | 3.0 | 1.0 | 5.7 | 0.0 | 4.2 |
| Branches | 12.0 | 10.9 | 15.5 | 4.0 | 5.8 | 2.3 | 5.1 | 17.4 | 10.5 | 16.0 | 13.6 | 14.7 | 19.2 | 11.3 |
| Invoke | 0.0 | 0.5 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.2 | 0.9 | 0.3 | 0.0 | 0.1 |
| Others | 0.0 | 1.0 | 0.6 | 0.2 | 2.5 | 2.4 | 9.4 | 0.0 | 7.1 | 4.7 | 2.2 | 4.2 | 5.9 | 3.1 |
| Total | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 | 100.0 |

STACK (bytes)

| Max. stack | 2 | 8 | 4 | 24 | 20 | 14 | 6 | 18 | 16 | 12 | 22 | 16 | 13.5 |
| Avg. stack | 2.08 | 2.37 | 2.14 | 11.76 | 6.30 | 6.77 | 3.36 | 1.89 | 2.73 | 3.15 | 2.19 | 4.83 | 3.08 | 4.1 |

PERFORMANCE OVERHEAD FOR ELLUL’S AOT (% of native C)

| Total | 663.9 | 693.2 | 475.8 | 250.4 | 226.4 | 124.3 | 122.8 | 492.5 | 272.6 | 376.0 | 790.8 | 210.2 | 205.6 | 373.7 |
| push/pop | 266.9 | 200.8 | 202.2 | 166.4 | 105.3 | 57.2 | 105.5 | 205.5 | 105.6 | 123.8 | 137.7 | 80.9 | 77.5 | 137.8 |
| load/store | 240.3 | 177.5 | 191.0 | 42.5 | 43.9 | 28.5 | 25.2 | 190.4 | 111.7 | 89.2 | 165.3 | 67.6 | 47.6 | 109.3 |
| mov(w) | 23.3 | 14.8 | 4.5 | 3.9 | 2.6 | 1.2 | 4.2 | 8.0 | 5.1 | 5.3 | 17.6 | 3.0 | 10.9 | 7.4 |
| other | 133.5 | 118.4 | 78.1 | 37.7 | 74.6 | 35.1 | 36.2 | 88.8 | 49.0 | 97.7 | 94.8 | 37.4 | 63.4 | 72.7 |
| vm | 0.0 | 181.7 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 60.0 | 375.4 | 27.3 | 2.1 | 50.1 |

OVERHEAD REDUCTION PER COMPILER OPTIMISATION (% of native C)

| Impr. peephole | -233.5 | -157.7 | -149.4 | -60.3 | -48.2 | -23.1 | -36.5 | -186.9 | -54.2 | -58.8 | -60.2 | -35.2 | -54.5 | -89.1 |
| Stack caching | -40.0 | -56.0 | -57.3 | -98.4 | -58.0 | -39.8 | -16.2 | -77.8 | 67.7 | -40.7 | -63.1 | -41.4 | -24.2 | -48.6 |
| Pop. val. caching | -133.1 | -84.9 | -67.4 | -6.8 | -12.9 | -8.8 | -10.7 | -51.0 | -28.8 | -24.5 | -41.5 | -15.4 | -15.5 | -38.5 |
| Mark loops | -102.9 | -46.8 | -85.4 | -5.0 | -10.9 | -8.0 | -7.9 | -114.9 | -18.0 | -40.0 | -54.3 | -38.2 | -28.6 | -42.4 |
| Const shift | -17.1 | -35.4 | -18.4 | -45.2 | -20.9 | -3.8 | 0.0 | -9.6 | -10.1 | 0.0 | -17.2 | -3.3 | -13.9 | -2.3 |
| 16-bit array index | -53.2 | -34.9 | -15.7 | -13.9 | -5.5 | -4.2 | -2.8 | -36.2 | -9.7 | -38.9 | -19.7 | -1.7 | -9.0 | -18.9 |
| SIMUL | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.1 | 0.4 | 4.9 |
| Lightw. methods | 0.0 | 207.3 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 36.6 | 0.0 | 0.0 |

PERFORMANCE OVERHEAD AFTER COMPILER OPTIMISATIONS (% of native C)

| Total | 101.2 | 88.5 | 65.2 | 57.6 | 45.7 | 19.5 | 17.7 | 75.7 | 84.6 | 58.9 | 156.3 | 30.5 | 70.2 | 67.0 |
| push/pop | 0.0 | -2.8 | 0.0 | 37.4 | 0.1 | 2.9 | 2.0 | -0.2 | -13.7 | 2.5 | 20.4 | 5.6 | 1.7 | 4.3 |
| load/store | 1.0 | 29.3 | 27.0 | -2.3 | 20.3 | 4.3 | 2.4 | 4.5 | 54.3 | 17.1 | 72.0 | 2.7 | 13.5 | 18.9 |
| mov(w) | 10.0 | 9.4 | 11.8 | 5.6 | 1.5 | 0.1 | 2.9 | 6.8 | 7.4 | 9.6 | 14.9 | 5.1 | 4.4 | 6.9 |
| other | 90.2 | 52.5 | 26.4 | 16.9 | 23.8 | 12.2 | 10.4 | 64.7 | 35.5 | 28.8 | 35.7 | 17.0 | 46.1 | 35.4 |
| vm | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.1 | 0.8 | 13.2 | 0.0 | 4.4 | 1.5 |
the large overhead of the interpreter would dwarf any differences between the different versions of the AOT compiler.

Using the simple optimiser, the types 1, 2, and 3 overhead are all significant, at 138%, 109%, and 73%, 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 the peephole optimiser is improved to include non-consecutive push/pop pairs, push/pop overhead drops by 100.2% (of native C performance), but if the push and pop target different registers, they are replaced by a move instruction, resulting in an increase of 11.5% in move overhead. For a 16-bit value, this takes one cycle (for a MOVW instruction) instead of eight cycles for two pushes and two pops. The increase in moves shows most of the extra cases that are handled by the improved optimiser are replaced by a move instead of eliminated, since the 11.5% extra move overhead matches a 92% reduction in push/pop overhead.

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

After eliminating the type 1 overhead almost completely, popped value caching is added to remove a large number of the unnecessary load instructions. This reduces the memory traffic significantly, as is clear from the reduced load/store overhead, while the other types remain stable. Adding the mark loops optimisation further reduces loads, and this time also stores, by pinning common variables to a register. But it uses slightly more move instructions, and the fact that fewer registers are available for stack caching means that stack values have to be spilled to memory more often. While 53.0% is saved on loads and stores, the push/pop and move overhead increase by 6.0% and 5.6%, respectively.

Most of the push/pop and load/store overhead has now been eliminated, and the type 3 overhead, unaffected by these optimisations, has become the most significant source of overhead. This type has many different causes, but half of it can be eliminated with three instruction set optimisations. These optimisations, especially the 16-bit array index, also reduce register pressure, resulting in slight decreases in the other overhead types, although these are minimal in comparison. The FFT and CoreMark benchmarks are the only ones to do 16-bit to 32-bit multiplication, so the average improvement for SIMUL is small, but Table 9 shows it is significant for these two benchmarks.

Combined, these optimisations reduce performance overhead from 377% to 67% of native C performance.

### 8.3 Code Size

Next we examine the effects of the 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 8 and 9, again split up per instruction category and benchmark, respectively. Table 10 shows the detailed per-benchmark data. For the first three optimisations, the two graphs follow a similar pattern as the performance graphs. These optimisations eliminate the need to emit certain instructions, which reduces code size and improves performance at the same time.

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

For each value that is live at the beginning of the loop, a load is emitted before the mark loops block, so in terms of code size there is only a benefit if it is loaded more than once. Code size may actually increase if the value is then popped destructively, since this would emit an extra mov.
### Table 10. Code Size Data Per Benchmark

| CODE SIZE (BYTES) | B.sort | H.sort | Bin.Search | XXTEA | MD5 | RC5 | FFT | Outlier | LEC | CoreMark | MoteTrack | HeatCalib | HeatDetect | average |
|-------------------|--------|--------|------------|-------|-----|-----|-----|--------|-----|----------|-----------|-----------|-----------|---------|
| Bytecode          | 105    | 198    | 120        | 462   | 3074| 516 | 575 | 411    | 402 | 3504     | 3050      | 406       | 3413      |         |
| Native C          | 118    | 298    | 146        | 1442  | 9458| 910 | 1292| 380    | 560 | 6128     | 3906      | 1944      | 5294      |         |
| AOT original      | 418    | 1012   | 412        | 3792  | 29502| 4090| 2576| 1402   | 1628| 13982    | 12784     | 2454      | 17248     |         |
| AOT optimised     | 258    | 596    | 310        | 2326  | 14654| 2018| 1324| 800    | 1056| 8990     | 8478      | 1610      | 10346     |         |

| CODE SIZE OVERHEAD FOR ELLUL’S AOT (% of native C) | Total | push/pop | load/store | mov(w) | other |
|-----------------------------------------------------|-------|----------|------------|--------|-------|
| Bytecode                                            | 254.2 | 71.2     | 88.1       | 10.2   | 84.7  |
| Native C                                            | 239.6 | 80.5     | 73.8       | 9.4    | 75.8  |
| AOT original                                        | 182.2 | 60.3     | 74.0       | 4.1    | 43.8  |
| AOT optimised                                       | 163.0 | 13.3     | 28.4       | 2.6    | 28.2  |

| OVERHEAD REDUCTION PER COMPILER OPTIMISATION (% of native C) | Total | push/pop | load/store | mov(w) | other |
|-------------------------------------------------------------|-------|----------|------------|--------|-------|
| Bytecode                                                   | −67.8 | 6.1      | 6.9        | 1.7    | −27.2 |
| Native C                                                   | −53.0 | 16.1     | 41.6       | 6.7    | −22.8 |
| AOT original                                               | −45.3 | 27.4     | 49.3       | 6.8    | −8.2  |
| AOT optimised                                              | −38.3 | 0.0      | 11.6       | −1.2   | −11.6 |

| CODE SIZE OVERHEAD AFTER COMPILER OPTIMISATIONS (% of native C) | Total | push/pop | load/store | mov(w) | other |
|----------------------------------------------------------------|-------|----------|------------|--------|-------|
| Bytecode                                                       | 118.6 | 100.0    | 112.3      | 0.0    | 59.3  |
| Native C                                                       | 100.0 | 23.7     | 27.4       | 1.7    | 53.5  |
| AOT original                                                   | 112.3 | 27.4     | 13.3       | −2.8   | 40.1  |
| AOT optimised                                                 | 54.9  | 3.0      | 62.1       | 2.5    | 18.4  |
Stores follow a similar argument. Also, for small methods the extra registers used may mean more call-saved registers have to be saved in the method prologue. Finally, the performance advantage is present in each runtime iteration but the effect on code size 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 five instructions. In most cases, the unrolled shifts will be smaller, but MD5 and XXTEA show a small increase in code size, since they contain 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 these when the array access instructions expect a 16-bit index, and the reduced work the access instruction needs to do, saves about 14% code size overhead in addition to the 19% 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.

### 8.3.1 VM Code Size and Break-even Point

These more complex code generation techniques do increase the size of the compiler. The first column in Table 11 shows the difference in code size between the AOT translator and Darjeeling’s interpreter. The basic AOT approach is 6,863 bytes larger than the interpreter, and each of the 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 the reduction in total size is shown,
as opposed to the overhead used elsewhere, to be able to calculate the break-even point. Using the improved peephole optimiser adds 276 bytes to the VM, but it reduces the size of the generated code by 14.6%. If more than 1.9kB is available to store user programmes, then 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 17.8kB.

As is often the case, there is a tradeoff between size and performance. The interpreter is smaller than each version of the AOT compiler, and Table 10 shows that bytecode is smaller than both native C and AOT compiled code, but the interpreter’s performance penalty may be unacceptable in many cases. AOT compilation 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 more code to be loaded onto a device.

For the smallest devices a subset of the optimisations could be used to limit the VM size and still get both a reasonable performance, and most of the code size reduction. For example, dropping the markloop optimisation reduces the size of the VM by 3KB but keeps most of the reduction in generated code size, while performance overhead increases to around 109%.

8.3.2 VM Memory Consumption. The last column in Table 11 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 25 bytes for a number of commonly used values such as a pointer to the next instruction to be compiled, the number of instructions in the method, and so on. The simple stack caching approach adds an 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 87 bytes of memory during the compilation process.

8.4 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 9 shows the distribution of the JVM instructions executed in each benchmark and both the maximum and average number of bytes on the JVM stack. There are some important differences between the benchmarks. While the sort benchmarks on the left are almost completely load/store bounded, XXTEA, RC5, and MD5 are much more computationally intensive, spending fewer instructions on loads and stores and more on math or bitwise operations. The left three benchmarks and the outlier detection benchmark only have a few bytes on the stack, but as the benchmarks contain more complex expressions, the number of values on the stack increases.

The second parts of Tables 9 and 10 first show the overhead before optimisation, split up in the four instruction categories. Then the effect of each optimisation on the total overhead is listed, followed by 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, search, and outlier detection, leaving less overhead to remove for stack caching. For benchmarks with more complicated expressions 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 computationally intensive benchmarks, the load/store overhead is much smaller, but the higher stack size means stack caching is very important for these benchmarks.

The smaller benchmarks highlight certain specific aspects of our approach, while the larger CoreMark benchmark covers a mix of different types of processing. As a result, it is an average case in almost every row in Table 9.

**Bubble sort** Next, we look at bubble sort in some more detail. After optimisation, we see that most of the stack-related overhead has been eliminated and of the 101.2% remaining performance overhead, most is due to other sources. For bubble sort there is a single, clearly identifiable source. 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 101.2% overhead, 93.1% is caused by these address calculations.

**HeatCalib and FFT** Table 10 shows that after optimisation the HeatCalib benchmark has a negative code size overhead. This is caused by the fact that the C versions are compiled using avr-gcc’s -O3 optimisations, optimising for performance instead of code size. In this case, as well as for FFT, this caused avr-gcc to duplicate a part of the code, which improves performance but at the cost of a significantly larger code size.

**MoteTrack** The MoteTrack benchmark is by far the slowest of our benchmarks, at a 156% overhead compared to native C. This is mostly due to its complex data structure. MoteTrack stores a database of reference RSSI signatures in Flash memory. In C, this is a complex struct containing a number of sub-structures and fixed-sized arrays. In Java, this becomes a collection of objects and arrays, containing many references.

Since the layout of the complete C structure is known at compile time, the C function to load a reference signature from the database can simply use memcpy to copy a block of 80 bytes from Flash memory to RAM. In Java, the method to read from Flash memory must follow several references to find the objects on the heap in which to store each value. As a result, reading a single signature takes 1,455 cycles in Java and only 735 cycles in C. After the reference signature is loaded, the fixed offsets in the C structure means using the loaded signature is also more efficient in C than in Java, which must again follow a number of references to reach the data.

**LEC** In Section 1, we calculated that the LEC compression algorithm reduced the energy spent to transmit the sample ECG data by 651μJ, at the expense of 246μJ spent on CPU cycles compressing the data, when implemented in C and using the ATmega128 CPU and CC2420 radio.

A compression algorithm like LEC is a good example of an optimisation that may be part of an application loaded onto a sensor node. However, if the overhead of using a VM is too high, then the cost of compression may outweigh the energy saved on transmission. Table 9 shows that using the baseline AOT approach, the LEC benchmark has an overhead of 272.6%, which means the CPU has to stay active longer, and compressing the data would cost 246μJ * 3.726 ≈ 917μJ, which is more than the 651μJ saved on transmission.

After applying all optimisations, the overhead is reduced to 84.6%, resulting in 246μJ * 1.846 ≈ 454μJ spent on compression. While the savings are less than when using native C to compress the data, the optimisations mean that in this scenario, we can save on transmission costs by using LEC compression, while using the baseline AOT approach, LEC compression results in a net loss.
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%, 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 memory.

For most benchmarks the maximum of seven register pairs to pin variables to was also the best option. At a lower average stack depth, the lower number of registers available for stack caching is easily compensated for by cheaper variable access. For XXTEA, however, the cost of spilling more stack values to memory outweighs the benefits of pinning more variables when too many variables are pinned. Figure 10 shows the overhead 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 five pinned register pairs, at which the total overhead is only 43%, instead of 58% at seven pinned register pairs.

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

Figure 11 shows the performance for each benchmark, as the number of pinned register pairs is increased. Of the smaller benchmarks, the four 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, RC5, and LEC. It should be possible to develop a simple heuristic using the maximum stack depth in a loop to allow the VM to make a better decision on the number of registers to pin. Since our current VM always pins seven pairs, we used this as our end result and leave this heuristic to future work.

8.5 Method Invocation

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

Table 12 lists the functions of the CoreMark, FFT, and heap sort benchmarks and the number of times they are called in a single run. Next, we list the way they are implemented in C. CoreMark is the most extensive benchmark. It only defines normal functions, which are inlined by avr-gcc in...
Table 12. Methods per Benchmark and Relative Performance for Normal, Lightweight Invocation, and Inlining

| Method          | # calls | Java Base version | Java Alternative version | Using normal method calls |
|-----------------|---------|-------------------|--------------------------|---------------------------|
| ee_isdigit      | 3920    | normal (inlined)  | manually inlined          | lightweight (handw.)      |
| core_state_transition | 1024   | normal            | lightweight               | lightweight                |
| circu8          | 584     | normal (inlined)  | manually inlined          | manually inlined           |
| circu16         | 292     | normal            | lightweight               | normal                     |
| calc_func       | 223     | normal            | lightweight               | normal                     |
| compare_idx     | 209     | normal (inlined)  | ProGuard inlined          | ProGuard inlined           |
| core_list_find  | 206     | normal            | lightweight               | normal                     |
| compare_complex | 110     | normal            | ProGuard inlined          | ProGuard inlined           |
| circu32         | 64      | normal            | lightweight               | normal                     |
| matrix_sum      | 16      | normal            | lightweight               | normal                     |
| others (<16 calls each) | 39 | normal | normal | normal |
| cycles          |         | 2,705,654         | 2,863,302                 | 3,855,242                  |
| overhead v native C | 58.9%  | 68.2%            | 126.4%                    |
| code size       | 8990    | 9006              | 9,328                     |

**FFT**

| Method          | # calls | Java Base version | Java Alternative version | Using normal method calls |
|-----------------|---------|-------------------|--------------------------|---------------------------|
| FIX_MPY         | 768     | marked inline     | manually inlined          | lightweight (handw.)      |
| cycles          |         | 179,692           | 215,020                  | 661,360                   |
| overhead v native C | 17.7%  | 40.8%            | 333.1%                    |
| code size       | 1,342   | 1,320             | 1,408                     |

**Heap sort**

| Method          | # calls | Java Base version | Java Alternative version | Using normal method calls |
|-----------------|---------|-------------------|--------------------------|---------------------------|
| SWAP            | 1642    | #define           | manually inlined          | manually inlined           |
| siftDown        | 383     | normal            | lightweight               | manually inlined           |
| cycles          |         | 208,239           | 184,071                  | 437,264                   |
| overhead v native C | 88.5%  | 66.6%            | 295.8%                    |
| code size       | 596     | 662               | 602                       |

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

three cases. FFT only contains a single function: FIX_MPY. This function is marked with the inline compiler hint, which was followed by avr-gcc. Finally, Heap sort contains a macro to swap two array elements and has two loops that both repeatedly call the siftDown function.

The Java base version column shows the way these functions are implemented in the Java versions of the benchmarks. C macros and the functions that are inlined by the C compiler are manually inlined in the Java version. The other functions are converted to Java methods, two of which were automatically inlined by ProGuard in the CoreMark benchmark, since they are only called from a single location. In heap sort, the siftDown method is called from two places, and larger than ProGuard’s size threshold, so it is not automatically inlined. The most commonly called methods are transformed to lightweight methods, simply by adding the @Lightweight annotation.

In the next two columns, these choices are varied slightly to examine the effect of lightweight methods. For the CoreMark benchmark, we first replace the inlined implementation of the most frequently called method with a lightweight version. ee_isdigit returns true if a char passed to it is between ’0’ and ’9’. Since this is a very trivial method, manually written bytecode was used
for this lightweight method that only uses the stack and has no local variables. This slowed down the benchmark by 6%, adding 157,648 cycles. Since the method is called 3920×, this corresponds to an overhead of about 40 cycles per call, which is on the high side for such a small method.

This is due to another source of overhead from using a lightweight method that is hard to quantify: The Boolean result of \texttt{ee\_isdigit} is used to decide an \texttt{if} statement. Using inlined code, the VM can directly branch on the result of the expression \((c >='0' \&\& c <='9')\), but the lightweight method first has to return a Boolean, which is then tested after the lightweight call returns.

The last column shows the performance without lightweight methods, when all methods, except the manually inlined \texttt{ee\_isdigit} and \texttt{crcu8}, have to be implemented as normal Java methods. This adds a total of 1,149,588 cycles, making it 1.42× slower than the version using lightweight methods. Spread over 1,825 calls, this means the average method invocation added over 630 cycles, which is within the range predicted in Section 7.

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

Finally, for the \texttt{heap sort} benchmark, we normally use a lightweight method for \texttt{siftDown}. The version where \texttt{siftDown} is inlined shows that the lightweight method call adds some overhead compared to the inlined version but much less than a normal method call would. However, while manually inlining it is possible, it is not an attractive option, since the \texttt{siftDown} method is much more complex than \texttt{FIX\_MPY} or \texttt{ee\_isdigit}.

In terms of code size, using normal methods take slightly more space than a lightweight method. Listing 3 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 \texttt{CoreMark}'s \texttt{ee\_isdigit}, the inlined code is slightly smaller than the call, but the \texttt{heap sort} benchmark shows that inlining larger methods can result in significantly larger code. Since \texttt{siftDown} is called from two places, duplicating it leads to a 11% increase in code size for the 16-bit version of \texttt{heap sort}. For the 32-bit version, where \texttt{siftDown} is relatively larger, this increases to 27%.

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

### 8.6 AOT Compilation Time

Finally, we evaluate the AOT compilation time. The VM was optimised for runtime performance of the generated code and for a small VM size rather than a fast compilation process. However, we believe the resulting compilation time, which is in the order of a few seconds, will be acceptable for many applications. Referring back to the examples mentioned in the Introduction, most users of the Amulet smart watch will probably find it acceptable to wait for a few seconds when loading a new application. Similarly, updating the feature extraction algorithm in Mercury only needs to be done once, after which the device will collect data for an extended period of time. For applications like this, the initial delay at load time will be of less importance and quickly compensated for by the faster runtime performance.

Table 13 shows the time spent on AOT compilation for each benchmark and the average number of bytes of bytecode processed per second. It also shows how this compilation time is distributed.
Table 13. AOT Compilation Time

|                      | B.sort | H.sort | Bin.Srch | XXTEA | MD5   | RC5   | FFT   | Outlier | LEC   | CoreMk | MoteTr. | HeatCal. | HeatDet. | average |
|----------------------|--------|--------|----------|--------|-------|-------|-------|---------|-------|--------|---------|----------|----------|---------|
| **ELLUL’S PEEPHOLE OPTIMISER** |        |        |          |        |       |       |       |         |       |        |         |          |          |         |
| Comp. time (seconds) | 0.33   | 0.70   | 0.33     | 2.19   | 13.22 | 2.14  | 1.60  | 1.05    | 1.06  | 9.85   | 7.70    | 1.65     | 9.07     |         |
| peephole optimiser  | 14%    | 21%    | 13%      | 51%    | 50%   | 47%   | 26%   | 12%     | 29%   | 28%    | 25%     | 31%      | 27%      | 29%     |
| writing to flash    | 77%    | 68%    | 78%      | 37%    | 35%   | 39%   | 62%   | 77%     | 59%   | 61%    | 62%     | 59%      | 58%      | 59%     |
| aot compilation     | 10%    | 11%    | 10%      | 12%    | 15%   | 13%   | 12%   | 10%     | 12%   | 11%    | 13%     | 11%      | 15%      | 12%     |
| Bytecode/s          | 319    | 281    | 368      | 211    | 232   | 241   | 359   | 391     | 379   | 356    | 396     | 247      | 376      | 320     |
|                      |        |        |          |        |       |       |       |         |       |        |         |          |          |         |
| **SIMPLE STACK CACHING** |        |        |          |        |       |       |       |         |       |        |         |          |          |         |
| Comp. time (seconds) | 0.27   | 0.51   | 0.28     | 0.80   | 4.18  | 0.84  | 1.04  | 0.87    | 0.68  | 6.47   | 5.36    | 1.03     | 5.85     |         |
| writing to flash    | 90%    | 88%    | 90%      | 75%    | 67%   | 75%   | 85%   | 89%     | 85%   | 86%    | 83%     | 85%      | 79%      | 83%     |
| aot compilation     | 10%    | 12%    | 10%      | 25%    | 33%   | 25%   | 15%   | 11%     | 15%   | 14%    | 17%     | 15%      | 21%      | 17%     |
| Bytecode/s          | 393    | 389    | 430      | 577    | 735   | 612   | 554   | 474     | 591   | 542    | 570     | 393      | 583      | 526     |
|                      |        |        |          |        |       |       |       |         |       |        |         |          |          |         |
| **FULLY OPTIMISED**  |        |        |          |        |       |       |       |         |       |        |         |          |          |         |
| Comp. time (seconds) | 0.27   | 0.46   | 0.29     | 0.82   | 4.05  | 0.79  | 0.95  | 0.84    | 0.65  | 6.40   | 4.90    | 0.98     | 5.34     |         |
| writing to flash    | 89%    | 86%    | 88%      | 72%    | 61%   | 72%   | 83%   | 88%     | 81%   | 83%    | 80%     | 83%      | 74%      | 80%     |
| aot compilation     | 11%    | 14%    | 12%      | 28%    | 39%   | 28%   | 17%   | 12%     | 19%   | 17%    | 20%     | 17%      | 26%      | 20%     |
| Bytecode/s          | 374    | 423    | 413      | 536    | 750   | 627   | 569   | 475     | 607   | 549    | 614     | 426      | 638      | 539     |
over three main tasks: the peephole optimiser in the baseline version, writing to Flash memory, and on the AOT compilation process itself. The optimised version of the VM takes 6.4s to compile CoreMark, the largest benchmark, and processes on average 539 bytes of bytecode per second. There are a number of opportunities to improve on this, but this would come at the expense of a larger VM or higher memory consumption during the compilation process.

Writing to Flash. The largest factor in the overall compilation time, regardless of which optimisations are used, is writing to Flash memory. A significant part of this cost is due to the way the VM handles branches. As discussed in Section 3.2.2, when the VM encounters a BRTARGET instruction, it records the address where the next instruction will be generated in a separate table in Flash memory. This requires two writes: first to flush the page with native code that is currently being built and then to write the address in the separate table.

Since the branch target addresses are not needed until the whole method is translated, the VM could maintain a small buffer to collect a number of addresses before writing them to Flash, thus reducing the number of Flash writes. However, this would increase both memory consumption and the size of the VM. Since we believe the current compilation time will be acceptable for many applications, we choose memory consumption and VM size over compilation performance.

Effect of AOT Compiler Optimisations. The baseline version in Table 13 uses a peephole optimiser similar to the one used in Ellul’s work [10]. This can take a considerable amount of time, since it parses the generated binary code while trying to identify push/pop pairs that can be eliminated. Because the peephole optimiser’s code cache is flushed at branches, for benchmarks that contain a relatively high number of branches like search and sort, the cost is limited as the optimiser works on small blocks of code. However, for the computationally intensive benchmarks the optimiser often has a large block of code that it repeatedly tries to optimise as the code is generated, and up to half of the compilation time may be spent on peephole optimisation.

Adding stack caching, but no other optimisations, removes the need for the peephole optimiser and managing the cache itself is a rather simple process. Stack caching also reduces the size of the generated code significantly, which lowers the cost of writing to Flash memory. The result is an increase in the bytes of bytecode processed per second from 320 to 526.

When all optimisations are turned on, the change in performance is limited. The state necessary to maintain for most optimisations, including mark loops, is still relatively simple. While processing this state still results in an increase in the time spent on AOT compilation, these optimisations further reduce the size of the generated code, which results in a reduction in the cost of writing to Flash. Thus, we see a modest shift from time spent on reprogramming Flash to time spent in the AOT compiler. This results in a slight loss in two benchmarks, but the average performance shows a small improvement, increasing from 526 to 539 bytes of bytecode processed per second.

Note that the 16-bit array index optimisation slightly reduces the size of the bytecode. This explains why for bubble sort, the same total compilation time results in a lower number of bytes processed per second.

While AOT compilation does introduce a slowdown of at most few seconds when compiling a large application like CoreMark, MoteTrack, or Heat detection, our optimisations reduce this slowdown compared to the baseline version. Compared to an interpreting VM, the break-even point at which this initial delay is compensated for by faster runtime performance depends on the compilation time and the slowdown incurred by the VM.

As a worst case, we consider MoteTrack, which has both a long compilation time of 4.9s, and the highest remaining slowdown compared to native C of 156.3% shown in Table 9. The fully
interpreted benchmarks in Table 1 show a slowdown of between $12 \times$ and $555 \times$. Again using the worst case, the AOT compiled MoteTrack would be $12/2.563 \approx 4.7 \times$ faster than the interpreter. This puts the break-even point at $4.9s/(4.7 - 1) \approx 1.3s$: If the application is active for more than 1.3s after AOT compilation, the total used CPU time, and thus energy consumption, is lower than using an interpreting VM. This indicates that in terms of energy consumption and performance, AOT compilation will quickly be beneficial for applications that can tolerate the initial compilation delay.

9 CONCLUSIONS AND FUTURE WORK

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 [10, 11] 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. On average, performance is $377\%$ slower than native C, and the code $198\%$ larger.

We presented a complete set of techniques to mitigate this code size overhead and to further improve performance. Their effectiveness was evaluated using a set of benchmarks, including the industry standard CoreMark benchmark, and a number of real-world sensor-node applications. Combined, these optimisations result in a compiler that produces code that is on average only $67\%$ slower and $77\%$ larger than optimised C.

These optimisations do increase the size of the VM, but the break-even point at which this is compensated for by the smaller code it generates is well within the range of programme memory typically available on a sensor node. 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. This contribution focuses 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, such as the approaches taken by nesC [15] and Amulet [18].

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 [21]. In this contribution, we presented a number of modifications to the bytecode format to make it better suited for use on a sensor node, 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 runtime, 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 [22] 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 [39, 45]. 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.

REFERENCES

[1] Bowen Alpern et al. 1999. Implementing Jalapeño in Java. In Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA’99).
[2] Anon. 2015. Heap sort at Rosetta Code. Retrieved October 24, 2017 from https://rosettacode.org/mw/index.php?title=Sorting_algorithms/Heapsort&oldid=214878.
[3] Faisal Aslam. 2011. Challenges and Solutions in the Design of a Java Virtual Machine for Resource Constrained Micro-controllers. Ph.D. Dissertation. University of Freiburg.
[4] Faisal Aslam et al. 2008. Introducing TakaTuka—A java virtualmachine for motes. In Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SENSYS’08).
[5] Rahul Balani et al. 2006. Multi-level software reconfiguration for sensor networks. In Proceedings of the ACM SIGBED International Conference on Embedded Software (EMSOFT’09).
[6] Niels Brouwers, Koen Langendoen, and Peter Corke. 2009. Darjeeling, a feature-rich VM for the resource poor. In Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SENSYS’09).
[7] Chipcon. 2004. 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver datasheet. http://www.ti.com/lit/ds/symlink/cc2420.pdf.
[8] The Embedded Microprocessor Benchmark Consortium. 2017. CoreMark 1.0. Retrieved October 24, 2017 from http://www.eembc.org/coremark.
[9] Alexandre Courbot, Gilles Grimaud, and Jean-Jacques Vandewalle. 2010. Efficient off-board deployment and customization of virtual machine-based embedded systems. ACM Trans. Embed. Comput. Syst. 9, 3 (2 2010).
[10] Joshua Ellul. 2012. Run-time Compilation Techniques for Wireless Sensor Networks. Ph.D. Dissertation. University of Southampton.
[11] Joshua Ellul and Kirk Martinez. 2010. Run-time compilation of bytecode in sensor networks. In Proceedings of the International Conference on Sensor Technologies and Applications (SENSORCOMM’10).
[12] M. Anton Ertl. 1995. Stack caching for interpreters. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’95).
[13] Leon Evers. 2010. Concise and Flexible Programming of Wireless Sensor Networks. Ph.D. Dissertation. University of Twente.
[14] Andreas Gal, Christian W. Probst, and Michael Franz. 2006. HotpathVM: An effective JIT compiler for resource-constrained devices. In Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE’06).
[15] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. 2003. The nesC language: A holistic approach to networked embedded systems. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’03).
[16] James Gosling et al. 2015. The Java Language Specification. Java SE 8 Edition.
[17] Till Harbaum. 2006. NanoVM. Retrieved October 24, 2017 from http://harbaum.org/till/nanovm/index.shtml.
[18] Josiah Hester, Travis Peters, Tianlong Yun, Ronald Peterson, Joseph Skinner, Bhargav Golla, Kevin Storer, Steven Hearndon, Kevin Freeman, Sarah Lord, Ryan Halter, David Kotz, and Jacob Sorber. 2016. Amulet: An energy-efficient, multi-application wearable platform. In Proceedings of the 14th International Conference on Embedded Networked Sensor Systems (SENSYS’16).
[19] Kirak Hong, Jiin Park, Sungho Kim, Taekhoon Kim, Hwangho Kim, Bernd Burgstaller, and Bernhard Scholz. 2012. TinyVM: An energy-efficient execution infrastructure for sensor networks. Softw. Pract. Exp. 42, 10 (Oct. 2012), 1193–1209.
[20] Kirak Hong, Jinn Park, Taekhoo Kim, Sungho Kim, Hwangho Kim, Yousun Ko, Jongtae Park, Bernd Burgstaller, and Bernhard Scholz. 2009. TinyVM, an efficient virtual machine infrastructure for sensor networks. In Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SENSYS’09).

[21] Cheng-Hsueh A. Hsieh, John C. Gyllenhaal, and Wen-mei W. Hwu. 1996. Java bytecode to native code translation: The caffeine prototype and preliminary results. In Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO’96).

[22] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. 2005. The implementation of Lua 5.0. J. Univ. Comput. Sci. 11, 7 (2005).

[23] Joel Koshy and Raju Pandey. 2005. VM*: Synthesizing scalable runtime environments for sensor networks. In ACM Conference on Embedded Networked Sensor Systems (SENSYS’05).

[24] Andreas Krall. 1998. Efficient JavaVM just-in-time compilation. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT’98).

[25] Benoît Latré, Pieter De Mil, Ingrid Moerman, Bart Dhoedt, Piet Demeester, and Niek Van Dierdonck. 2006. Throughput and delay analysis of unslotted IEEE 802.15.4. J. Netw. 1, 1 (May 2006), 20–28.

[26] Philip Levis and David Culler. 2002. Maté: A tiny virtual machine for sensor networks. In Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SENSYS’05).

[27] Philip Levis, David Gay, and David Culler. 2005. Activesensor networks. In Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI’05).

[28] Philip Levis and David Culler. 2002. Maté: A tiny virtual machine for sensor networks. In Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SENSYS’05).

[29] Team libtom. 2017. LibTomCrypt. Retrieved October 24, 2017 from http://www.libtom.net/LibTomCrypt.

[30] Konrad Lorincz and Matt Welsh. 2006. MoteTrack: A Robust, Decentralized Approach to RF-Based Location Tracking. Retrieved from http://www.eecs.harvard.edu/konrad/projects/motetrack.

[31] Konrad Lorincz and Matt Welsh. 2007. MoteTrack: A robust, decentralized approach to RF-based location tracking. Pers. Ubiq. Comput. 11, 6 (Aug. 2007).

[32] Konrad Lorincz et al. 2009. Mercury: A wearable sensor network platform for high-fidelity motion analysis. In Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SENSYS’09).

[33] F. Marcelloni and M. Vecchio. 2009. An efficient lossless compression algorithm for tiny nodes of monitoring wireless sensor networks. Comput. J. 52, 8 (Nov. 2009).

[34] Microchip Technology. 2011. 8-bit Atmel Microcontroller with 128KBytes In-System Programmable Flash ATMega128/ATmega128L datasheet. http://ww1.microchip.com/downloads/en/DeviceDoc/doc2467.pdf.

[35] René Müller, Gustavo Alonso, and Donald Kossmann. 2007. A virtual machine for sensor networks. In Proceedings of the European Conference on Computer Systems (EuroSys’07).

[36] Gilles Muller et al. 1997. Harissa: A flexible and efficient java environment mixing bytecode and compiled code. In Proceedings of the Conference on Object-Oriented Technologies and Systems (COOTS’97).

[37] Oracle. 2005. Connected Limited Device Configuration. Retrieved October 24, 2017 from http://www.oracle.com/technetwork/java/cldc-141990.html.

[38] PhysioNet. 2018. PTB Diagnostic ECG Database ptbdb. 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. Retrieved from https://www.physionet.org/cgi-bin/atm/ATM.

[39] Yunhe Shi et al. 2005. Virtual machine showdown: Stack versus registers. In Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE’05).

[40] Guard Square. 2016. ProGuard 5.3.2. Retrieved Oct 24, 2017 from https://sourceforge.net/projects/proguard/.

[41] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. 2005. Avrora: Scalable sensor network simulation with precise timing. In Proceedings of the International Symposium on Information Processing in Sensor Networks (IPSN’05).

[42] UCLA Networked & Embedded Systems Laboratory. 2009. SOS operating system. Retrieved from https://github.com/nesl/sos-2x.

[43] David Wheeler and Roger Needham. 1998. XXTEA: Correction to XTEA. Technical Report. Computer Laboratory, University of Cambridge.

[44] Ingwar Wirjawan et al. 2008. Balancing computation and communication costs: The case for hybrid execution in sensor networks. Ad Hoc Netw. 6, 8 (Nov. 2008).

[45] Yuan Zhang et al. 2012. Swift: A register-based JIT compiler for embedded JVMs. ACM SIGPLAN Not. 47, 7 (Jan 2012).