Ahead-of-Time Compilation of Stack-Based JVM Bytecode on Resource-Constrained Devices

Niels Reijers, Chi-Sheng Shih
Embedded Systems and Wireless Networking Lab
Department of Computer Science and Information Engineering
National Taiwan University

Abstract

Many virtual machines have been developed for tiny devices with only a few KB of RAM and tens to a few hundred KB of flash memory. They pack an impressive set of features in a very limited space, but suffer from a large performance penalty: a slowdown of one to two orders of magnitude compared to optimised native code is typical for most VMs, reducing throughput and increasing power consumption.

Compiling the VM's bytecode to native code to improve performance has been studied extensively for larger devices, but has not received much attention in the context of sensor networks, where the restricted resources mean most modern techniques cannot be applied. Simply replacing each VM instruction with a predefined sequence of native instructions is known to improve performance, but there is still a large gap to native C performance and the trade off is that the resulting code is several times larger than the compiled C equivalent, limiting the amount of code that can be loaded onto a device.

This paper presents techniques to mitigate this code size overhead, making translation to native code a more attractive option, and to further improve performance. We start by analysing the overhead resulting from the basic approach using a set of benchmarks with different characteristics: sort, binary search, fft, rc5, xxtea and md5. We identify three distinct sources of overhead, two of which are related to the JVM's stack-based architecture, and propose a set of optimisations to target each of them.

Combined these optimisations reduce code size overhead by 59%. While they do increase the size of the VM, the break even point at which this fixed cost is compensated for is well within the range of memory typically available on a sensor device, allowing us to load more code on a device. Performance overhead is reduced by 79%, resulting in an average performance that is only 68% slower than optimised C.

International Conference on Embedded Wireless Systems and Networks (EWSN) 2017
20–22 February, Uppsala, Sweden

ISBN: 978-0-9949886-1-4

1 Introduction

Sensor nodes and other 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: much 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 we can use normal operating systems, languages, and compilers, but in this paper, 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 several advantages to running a VM. One is ease of programming. Many VMs allow the developer to write programmes at a higher level of abstraction than the bare-metal C programming that is still common for these devices. Second, a VM can offer a safe execution environment, preventing buggy or malicious code from disabling the device. A third 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 all 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. Often, 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 slowdown incurred by a VM means the CPU has to be active 10 to 100 times longer, this may suddenly become the dominant factor.

As an example, one of the few applications reporting a detailed breakdown of its power consumption is Mercury [22], a platform for motion analysis. The greatest energy consumer is the sampling of a gyroscope, at 53.163 mJ. Only 1.664 mJ is spent in the CPU on application code for an activity recognition filter and feature extraction. When multiplied by 10 or 100 however, this becomes a very significant, or even by far the largest energy consumer. A more complex operation such as a 512 point FFT costs 12.920 mJ. For tasks like this, even a slowdown by a much smaller factor will have a significant impact on the total energy consumption.

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. 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 native 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 CPU used in the network. For the node it will be harder to provide a safe execution environment when it receives binary code.

Therefore, we focus on the middle option in this paper: translating the bytecode to native code on the device itself, at load time. The main research question we wish to answer is what tradeoffs are involved in AOT compilation on a sensor node, and how close an AOT compiling sensor node VM can come to native C performance.

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 [21] which provide specialised instructions for a specific problem domain. One of the first VMs proposed for sensor networks, Maté [20], 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 users. SwissQM [23] 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* [18] 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 [14, 7, 5, 12]. The smallest official Java standard is the Connected Device Limited Configuration [1], 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. TinyVM [15] reports a slowdown between 14x and 72x compared to native C, for a set of 9 benchmarks. DVM [6] has different versions of the same benchmark, where the fully interpreted version is 108x slower than the fully native version. Ellul reports some measurements on the TakaTuka VM [5, 9] where the VM is 230x slower than native code, and consumes 150x as much energy. SensorScheme [12] is up to 105x slower. Finally, Darjeeling [7] reports between 30x and 113x slowdown. 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 [24]. 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 [8]. Although the initialised objects are translated to C structures that are compiled and linked into a single image, the bytecode is still interpreted. While in general we can produce higher quality code when compiling offline, doing so sacrifices key advantages of using a VM.

Hsieh et al. describe an early ahead-of-time compiling desktop Java VM [16], focussing on translating the JVM's stack-based architecture to registers. In the Japaleño VM, Alpern et al. take an approach that holds somewhere between AOT and JIT compilation [3]. The VM compiles all code to native code before execution, but contains 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 light weight as possible, for example [19]. More recently these efforts have included JIT compilers targeted specifically at embedded devices. Swift [29] is a light-weight JVM that improves performance by translating a register-based bytecode to native code. But while the Android devices targeted by Swift may be considered embedded devices, they are still quite powerful and the transformations Swift does are too complex for the ATmega class of devices. HotPathVM [13] has lower requirements, but at 150KB for both code and data, this is still an order of magnitude above our target devices.

Given our extreme size constraints - ideally we only want to use in the order of 100 bytes of RAM to allow our approach to be useful on a broad range of devices, and leave ample space for other tasks on the device - almost all AOT and JIT techniques found in literature require too much resources. Indeed, some authors suggest sensor nodes are too restricted to make AOT or JIT compilation feasible [4, 28].

On the desktop, VM performance has been studied extensively, but for sensor node VMs this aspect has been mostly ignored. To the best of our knowledge AOT compilation on a sensor node has only been tried by Ellul and Martinez [10], 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 produces code that is 327% slower and 215% larger than optimised native C. While the reduced throughput may be acceptable for some applications, there are two other reasons why it is important to improve on these results: the loss of performance results in an equivalent increase in cpu power consumption, thus reducing battery life. More importantly, 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 paper makes the following contributions:

- We reveal three distinct causes of overhead in Ellul's approach of directly mapping JVM instructions to predefined fragments of native code.
- Using the results of this analysis, we propose five optimisations to address these sources of overhead.
- We reduce the code size overhead by 59%, and show that the increase in VM size is quickly compensated for, thus mitigating a drawback of the previous AOT approach.
- We eliminate most of the performance overhead caused by the JVM's stack-based architecture, and about 79% of overhead overall, leading to an average slowdown of only 1.7x compared to native C.

3 Ahead-of-Time translation

Our implementation is based on Darjeeling [7], 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 [9], it depends on the ability to execute code from RAM, which ATmega CPUs 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 then transformed into Darjeeling's own format, called an 'infusion'. For details of this transformation we refer to the Darjeeling paper [7]. Here it is sufficient to note 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.

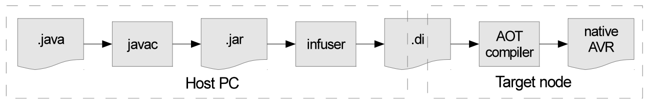

Figure 1. Java to native AVR compilation

3.1 Goals and limitations

Working on resource-constrained devices means we will have to make some compromises to improve performance. We would like our VM to fit as many scenarios as possible. In some cases multiple applications may be running on a single device. When new code is being loaded, the impact on concurrently running code should be as small as possible.

Therefore, our translation process should be very light weight. Specifically, we should use as little memory as possible since memory is a very 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 process each message in order, and free it after processing. The actual transmission protocol may still decide to keep more messages in memory, for example to handle out of order delivery efficiently, but we note that our translation process does not require it to do so.

This means we limit ourselves to a single pass, processing instructions one at a time, and keeping only small, fixed-size data structures in memory during the process. The only second pass we do is one to fill in addresses left blank by branch instructions, since we cannot know the target address of forward branches until the target instruction is generated.

The two points we compromise on are load time and code size. Compiling to native code does take 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. The native code produced 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 mitigating a 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 because after compilation, we lose the interpreter loop as a convenient place to switch between threads.

3.2 Translating bytecode to native code

The basic approach to translate bytecode to native code was first described by Ellul and Martinez [10]. When we receive a bytecode instruction, it is replaced 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 which 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 all our optimisations. The second column shows the code the AOT compiler will execute for each JVM instruction. Together, the first and second column match the case labels and body of a big

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

JVM	AOT compiler	AVR	cycles
0: BRTARGET(0)	*« record current addr »*		
1: SLOAD_0	emit_LDD(R1,Y+0)	LDD R1,Y+0	4
	emit_PUSH(R1)	PUSH R1	4
2: SCONST_1	emit_LDI(R1,1)	LDI R1,1	2
	emit_PUSH(R1)	MOV R2,R1	1
3: SUSHR	emit_POP(R2)		
	emit_POP(R1)	POP R1	4
	emit_RJMP(+2)	RJMP +2	2
	emit_LSR(R1)	LSR R1	2
	emit_DEC(R2)	DEC R2	2
	emit_BRPL(-2)	BRPL -2	3
	emit_PUSH(R1)		
4: SSTORE_0	emit_POP(R1)		
	emit_STD(Y+0,R1)	STD Y+0,R1	4
5: SLOAD_0	emit_LDD(R1,Y+0)	LDD R1,Y+0	4
	emit_PUSH(R1)	PUSH R1	4
6: SLOAD_1	emit_LDD(R1,Y+2)	LDD R1,Y+2	4
	emit_PUSH(R1)		
7: IF_SCMPGT 0:	emit_POP(R1)		
	emit_POP(R2)	POP R2	4
	emit_CP(R1,R2)	CP R1,R2	2
	emit_branchtag(GT,0)	BRGT 0:	1 or 2

switch statement in our compiler. The third column shows the resulting AVR native code, which is currently almost a 1-on-1 mapping, with the exception of some small optimisations by a simple peephole optimiser described below.

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

Peephole optimisation From Table 1 it is clear that this approach results in many unnecessary push and pop instructions. Each instruction must get 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 [9]. The compilation process results in many push instructions that are immediately followed by a pop. If these target the same register, they have no effect and are removed. If the source and destination registers differ, the two instructions are replaced by a move. The result is shown in the third column of Table 1. Two push/pop pairs have been removed, and one has been replaced by a move.

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

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 we encounter a BRTARGET during compilation, we do not emit any code, but record the address where the next instruction will be emitted in a separate part of flash. When we encounter a branch instruction, we emit a temporary 3-word 'branch tag' instead, containing the BRTARGET id and the branch condition. After code generation is finished and all target addresses are known, we scan the code again to replace each branch tag with the real branch instruction.

There is still the matter of the different sizes a branch may take. We could simply add NOPs to smaller branches to keep the size of each branch at 3 words, but this causes a performance penalty on small, non-taken branches. Instead, we do another scan of the code, before replacing the branch tags, and update 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 also result in smaller generated code.

4 Three 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 root causes of this overhead. We find three main sources, where the first two are a direct result of the JVM's stack-based architecture.

Type 1: Pushing and popping values The compilation process initially results in a large number of push and pop instructions. In our simple example 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.

Type 2: Loading and storing values A second source of overhead is also due to the JVM's stack-based architecture. Each operation consumes its operands from the stack, but often the same value is needed again soon after. In this case, because the value is no longer on the stack, we need to do another load, which will result in another read from memory.

In our example, it is clear that the SLOAD_0 instruction at label 5 is unnecessary since the value is already in R1.

Type 3: Bytecode limitations A final source of overhead comes from optimisations that are done in native code, but are not possible in JVM bytecode. The JVM instruction set is very simple, which makes it easy to implement, but this also means some things cannot be expressed as efficiently. Given enough processing power, compilers can do the complex transformations necessary to make the compiled JVM code run almost as fast as native C, but on a sensor node we do not have such resources and must simply execute the instructions as they are.

In our example we see that there is no way to express a single bit shift directly. Instead we have to load the constant 1 onto the stack and execute the generic bit shift instruction. Compare this to addition, where the JVM bytecode does have a special INC instruction to add a constant value to a local variable. Another example is iterating over an array. In JVM

Table 2. Simple stack caching

JVM	AOT compiler	AVR	cycles	cache state R1	cache state R2	cache state R3
0: BRTARGET(0)	*« record current addr »*					
1: SLOAD_0	operand_1 = sc_getfreereg()					
	emit_LDD(operand_1,Y+0)	LDD R1,Y+0	4			
	sc_push(operand_1)			Int1		
2: SCONST_1	*« skip codegen »*			Int1		
3: SUSHR	*« skip codegen, emit special case »*			Int1		
	operand_1 = sc_pop()					
	emit_LSR(operand_1)	LSR R1	2			
	sc_push(operand_1)			Int1		
4: SSTORE_0	operand_1 = sc_pop()					
	emit_STD(Y+0,operand_1)	STD Y+0,R1	4			
5: SLOAD_0	operand_1 = sc_getfreereg()					
	emit_LDD(operand_1,Y+0)	LDD R1,Y+0	4			
	sc_push(operand_1)			Int1		
6: SLOAD_1	operand_1 = sc_getfreereg()			Int1		
	emit_LDD(operand_1,Y+2)	LDD R2,Y+2	4	Int1		
	sc_push(operand_1)			Int2	Int1	
7: IF_SCMPGT 0:	operand_1 = sc_pop()			Int1		
	operand_2 = sc_pop()					
	emit_CP(operand_1, operand_2);	CP R2,R1	2			
	emit_branchtag(GT, 0);	BRGT 0:	1 or 2			

bytecode, each array access will consume the array reference and index. We have to load them onto the stack again and redo the address calculation for each iteration, while native could would typically just slide a pointer over the array.

5 Optimisations

We now introduce several optimisations, targeting these three types of overhead in order.

5.1 Improving the peephole optimiser

Our first optimisation is a small but effective extension to the simple peephole optimiser. Instead of only optimising consecutive push/pop pairs, we can optimise any pair if the *target* register of the pop is not used in between the push and the pop. Two push/pop pairs remain in Table 1. 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.

This optimisation adds 264 bytes to the optimiser, since it now needs to understand the bytecode well enough to determine which registers are being used.

5.2 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 [11] to eliminate most of the remaining push/pop overhead. While not a new technique, the tradeoffs are 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 simple or the overhead from managing it will outweigh the time saved by reducing memory accesses. Since we only use the cache state at load time, we can afford to spend more time on it. Second, the simplicity of the approach means it requires very little memory: only 11 bytes of RAM and just over 1KB of code more than the peephole optimiser.

The idea 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, and we encounter an ADD instruction, we do not need to access main memory, but can simply add these registers, and update the cache state. We only push values to the real stack when we run out of registers.

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 will need a free register to load the value into, which will later be pushed onto the stack. If all registers are in use, `getfree` spills the register that's lowest on the stack to memory by emitting a PUSH, and then returns that register. This way the top of the stack is kept in registers, while lower elements may be spilled to memory.
- `pop`: Pops the top element off the stack and tells the code generator in which register to find it. If stack elements have previously been spilled to main memory and no elements are left in registers, `pop` will emit a real POP instruction to get the value back from memory.
- `push`: Simply 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 2 we translate the same example we used before, but this time using stack caching. To save space, Table 2 also includes the constant shift optimisation described in Section 5.5. The `emit_PUSH` and `emit_POP` instructions have been replaced by calls to the cache manager, and instructions that

Table 3. Popped value caching

JVM	AOT compiler	AVR	cycles	cache state R1	cache state R2	cache state R3
0: BRTARGET(0)	*« record current addr »*					
1: SLOAD_0	operand_1 = sc_getfreereg()					
	emit_LDD(operand_1,Y+0)	LDD R1,Y+0	4			
	sc_push(operand_1)			Int1 LS0		
2: SCONST_1	*« skip codegen »*			Int1 LS0		
3: SUSHR	*« skip codegen, emit special case »*			Int1 LS0		
	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()			LS0		
	emit_STD(Y+0,operand_1)	STD Y+0,R1	4	LS0		
5: SLOAD_0	*« skip codegen, just update cache state »*			Int1 LS0		
6: SLOAD_1	operand_1 = sc_getfreereg()			Int1 LS0		
	emit_LDD(operand_1,Y+2)	LDD R2,Y+2	4	Int1 LS0		
	sc_push(operand_1)			Int2 LS0	Int1 LS1	
7: IF_SCMPGT 0:	operand_1 = sc_pop_nondestructive()			Int1 LS0	LS1	
	operand_2 = sc_pop_nondestructive()			LS0	LS1	
	emit_CP(operand_1, operand_2);	CP R2,R1	2	LS0	LS1	
	emit_branchtag(GT, 0);	BRGT 0:	1 or 2	LS0	LS1	

load something on the stack start by asking the cache manager for a free register. The state of the stack cache is shown in the three columns added to the right. Currently it only tracks whether a register is on the stack or not. "Int1" marks the top element, followed by "Int2", etc. In the next two optimisations we will extend the cache state further.

The example only shows three registers, but in reality the ATmega128 has 32 8-bit registers. Since Darjeeling uses a 16-bit stack, we manage them as pairs. 10 registers are reserved, for example as a scratch register or to store a pointer to local or static variables, leaving 11 pairs available for stack caching.

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

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

5.3 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.

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

We add a function, `sc_can_skip`, to the cache manager. This function will examine the type of each instruction, its value tag, and the cache state. If it finds that we are loading a value that is already present in a register, it just 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 3 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, the `pop` has been changed to `pop_destructive`. This tells the cache manager that the value in the register will be destroyed, so the value tag has to be cleared again since R1 will no longer contain LS0. The SSTORE_0 instruction now calls `pop_tostore` instead of `pop`, to inform the cache manager it will store this value in the variable identified by SSTORE_0's value tag. This means the register once again contains LS0. If any other register was marked as containing LS0, the cache manager would clear that tag, since it is no longer accurate after we update the variable.

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

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

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

Table 4. Mark loops

JVM	AOT compiler	AVR	cycles	cache state R1	cache state R2	cache state R3
0: MARKLOOP(0,1)	*« emit markloop prologue:*	LDD R1,Y+0	4	LS0 PINNED		
	LS0 and LS1 are live »	LDD R2,Y+2	4	LS0 PINNED	LS1 PINNED	
1: BRTARGET(0)	*« record current addr »*			LS0 PINNED	LS1 PINNED	
2: SLOAD_0	*« skip codegen, just update cache state »*			Int1 LS0 PINNED	LS1 PINNED	
3: SCONST_1	*« skip codegen »*			Int1 LS0 PINNED	LS1 PINNED	
4: SUSHR	*« skip codegen, emit special case »*			Int1 LS0 PINNED	LS1 PINNED	
	operand_1 = sc_pop_destructive()	MOV R3,R1	1	LS0 PINNED	LS1 PINNED	
	emit_LSR(operand_1)	LSR R3	2	LS0 PINNED	LS1 PINNED	
	sc_push(operand_1)			LS0 PINNED	LS1 PINNED	Int1
5: SSTORE_0	*« skip codegen, move to pinned reg »*	MOV R1,R3	1	LS0 PINNED	LS1 PINNED	
6: SLOAD_0	*« skip codegen, just update cache state »*			Int1 LS0 PINNED	LS1 PINNED	
7: SLOAD_1	*« skip codegen, just update cache state »*			Int2 LS0 PINNED	Int1 LS1 PINNED	
8: IF_SCMPGT 0:	operand_1 = sc_pop_nondestructive()			Int1 LS0 PINNED	LS1 PINNED	
	operand_2 = sc_pop_nondestructive()			LS0 PINNED	LS1 PINNED	
	emit_CP(operand_1, operand_2);	CP R2,R1	2	LS0 PINNED	LS1 PINNED	
	emit_branchtag(GT, 0);	BRGT 1:	1 or 2	LS0 PINNED	LS1 PINNED	
9: MARKLOOP(end)	*« emit markloop epilogue: LS0 is live »*	STD Y+0,R1	4	LS0	LS1	

5.4 Mark loops

Popped value caching reduces the type 2 overhead significantly, but the fact that we have to clear the value tags at branch targets means that a large part of that overhead still remains. This is particularly true for loops, since each iteration will often use the same variables, but the branch to start the next iteration clears those values from the stack cache. This is addressed by the next optimisation.

Again, we modify the infuser to add a new instruction to the bytecode: MARKLOOP. This instruction is used to mark the beginning and end of each 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 they appear, sorted in descending order.

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

Because the only way into and out of the loop is through the MARKLOOP instructions, the values can remain pinned for the whole duration of the block, regardless of the branches made inside. This lets us eliminate more load instructions, and also replace store instructions by a much cheaper move to the pinned register. INC instructions, which increment a local variable, operate directly on the pinned register, saving both a load and a store. All these cases are handled in `sc_can_skip`, bypassing the normal code generation. We also need to make a small change to `sc_pop_destructive`. If the register we're about to pop is pinned, we cannot just return it since it would corrupt the value of the pinned local variable. Instead we will first emit a move to a free, non-pinned register, and return that instead.

In Table 4 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.

We then load LS0. Since it is pinned to register 1, we do not generate any code but only update the cache state. Next, SUSHR pops destructively. We cannot simply return register 1 since that would corrupt the value of variable LS0, so `sc_pop_destructive` emits a move to a free register and returns that register instead. Since LS0 is pinned, we can also skip SSTORE_0, but we do need to emit a move back to the pinned register.

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

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

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 big improvement from the 48 cycles spent in the original version in Table 1.

5.5 Constant bit shifts

Finally, we introduce an optimisation that targets the type 3 overhead. This kind of overhead is the hardest to address because it requires more complex transformations that usually take more resources than we can afford on a tiny device. Also, this type of overhead covers many different cases, and optimisations that help in a specific case may not be general enough to justify spending additional resources on it.

There is one case that is both common and easy to optimise: shifts by a constant number of bits appear in six of the seven benchmarks described in Section 6. They appear not only in computation intensive benchmarks, but also as optimised multiplications or divisions by a power of 2, which are

Table 5. Performance data per benchmark

BENCHMARK	b.sort	h.sort	b.srch	fft	xxtea	md5	rc5	average
EXECUTED JVM INSTRUCTIONS (%)								
Load/store	79.7	71.0	58.8	57.6	51.0	43.9	39.7	57.4
Constant load	0.2	8.1	9.8	10.8	12.5	18.9	18.8	11.3
Processing	8.0	7.9	13.1	22.7	32.4	29.1	37.3	21.5
math	8.0	5.6	9.2	12.0	10.1	12.4	11.3	9.8
bit shifts	0.0	2.3	3.9	7.5	8.1	5.4	7.7	5.0
bit logic	0.0	0.0	0.0	3.2	14.2	11.3	18.3	6.7
Branches	12.0	11.1	17.6	4.2	4.0	5.8	2.2	8.1
Others	0.0	2.0	0.7	4.5	0.1	2.5	2.2	1.7
STACK								
Max. stack (bytes)	8.0	8.0	8.0	8.0	24.0	20.0	14.0	12.9
Avg. stack (bytes)	2.6	3.0	2.8	3.0	11.8	6.3	6.8	5.2
PERFORMANCE OVERHEAD BEFORE OPTIMISATIONS (%)								
Total	398.6	419.3	416.0	479.0	239.5	213.3	120.1	326.5
push/pop	144.3	185.8	191.7	199.9	154.5	99.6	60.4	148.0
mov(w)	-6.0	-5.4	-5.8	-3.3	-1.2	-0.7	-2.4	-3.5
load/store	192.2	181.6	180.1	132.3	44.7	43.9	28.5	114.8
other	68.2	57.3	50.0	150.1	41.5	70.6	33.6	67.3
PERFORMANCE OVERHEAD REDUCTION PER OPTIMISATION (%)								
Impr. peephole	-126.2	-118.7	-133.0	-96.0	-51.6	-49.7	-26.7	-85.9
Stack caching	-10.0	-63.7	-56.6	-108.5	-98.8	-49.4	-34.2	-60.2
Pop. val. caching	-111.9	-88.0	-29.7	-50.8	-9.7	-14.2	-8.1	-44.6
Mark loops	-63.8	-45.3	-110.3	-47.2	+ 11.7	-8.8	-10.9	-39.3
Const shift	0.0	-10.6	-22.3	-77.9	-24.2	-44.6	-19.8	-28.5
PERFORMANCE OVERHEAD AFTER OPTIMISATIONS (%)								
Total	86.7	93.0	64.1	98.6	66.9	46.6	20.4	68.0
push/pop	0.0	0.0	0.0	0.0	39.4	0.1	2.9	6.1
mov(w)	10.0	7.5	11.2	4.6	4.7	2.1	0.5	5.8
load/store	8.5	40.1	28.1	21.7	-2.3	20.1	4.3	17.2
other	68.2	45.3	24.8	72.1	25.1	24.4	12.7	38.9

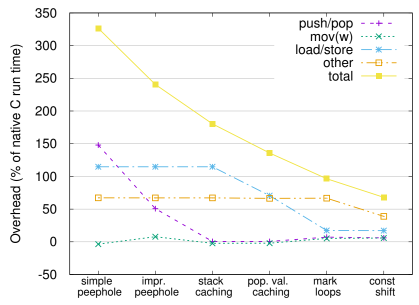

Figure 2. Perf. overhead per category

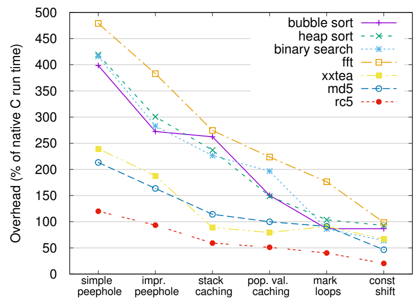

Figure 3. Perf. overhead per benchmark

common in many programmes.

In JVM bytecode the shift operators take two operands from the stack: the value to shift, and the number of bits to shift by. While this is generic, it is not efficient for constant shifts: we first need to push the constant onto the stack, and then the bit shift is implemented as a simple loop which shifts one bit at a time. If we already know the number of bits to shift by, we can generate more efficient code. When we encounter a constant push followed by a bit shift, we skip the code generation for both instructions, and emit a special case instead. This only adds 594 bytes to our VM, but it improves performance, sometimes very significantly, for all but one of our benchmarks.

For arithmetic operations with a constant operand, our translation process results in loading the constant and performing the operation, similar to what avr-gcc produces in most cases. But for constant shifts, the loop adds significant overhead, which is why we optimise it as a special case.

Surprisingly, when we first implemented this, one benchmark performed better than native C. It turns out avr-gcc does not optimise constant shifts in all cases. Since our goal is to examine how close a sensor node VM can come to native performance, it would be unfair to include an optimisation that is not found in the native compiler, but could easily be added. We implemented a version that is close to what avr-gcc does, but never better. We only consider cases optimised by avr-gcc. For these, we first emit whole byte moves if the number of bits to shift by is 8 or more, followed by single bit shifts for the remainder. As mentioned before, this optimisation was already included in the example from Table 2 on, so the effect can be seen by comparing the SCONST_1 and SUSHR instructions in tables 1 and 2.

6 Evaluation

We use a set of seven different benchmarks to measure the effect of our optimisations: *bubble sort*: from the Darjeeling sources, and used in [7, 9], *heap sort*: standard heap sort, *binary search*: taken from the TakaTuka [5] source code, *fft*: fixed point FFT, adapted from the widespread fix_fft.c, *xxtea*: as published in [27], *md5*: also from the Darjeeling sources, and used in [7, 9], and *rc5*: from LibTomCrypt [2]. For each, we implemented both a C and a Java version, keeping both implementations as close as possible. Since javac does very few optimisations, we manually optimised the code in some places. The same optimisations did not affect the C version, indicating avr-gcc already does similar transformations on the original code. We use javac version 1.7.0 and avr-gcc version 4.9.1. The C benchmarks are compiled at optimisation level -O3, the rest of the VM at -Os.

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

We run our VM in the cycle-accurate Avrora simulator [26], emulating an ATmega128 processor. We modified Avrora to get detailed traces of the compilation process and of the runtime performance of both C and AOT compiled code. While we implemented our approach for AVR, it can be applied to any CPU in this class that has enough registers for stack caching, such as Cortex M0 or the MSP430 used in Ellul's work [9].

Our main measurement for both code size and performance is the overhead compared to optimised native C. To

Table 6. Code size data per benchmark

BENCHMARK	b.sort	h.sort	b.srch	fft	xxtea	md5	rc5	average
CODE SIZE (BYTES)								
JVM	105	293	127	607	456	3060	514	
Native C	168	534	188	1214	1238	9458	910	
AOT original	530	1660	658	2512	3568	28762	3958	
AOT optimised	352	932	462	1490	2250	14548	2130	
CODE SIZE OVERHEAD BEFORE OPTIMISATIONS (%)								
Total	211.8	209.7	250.0	106.9	187.7	204.1	334.9	215.0
push/pop	82.4	88.1	102.1	59.0	114.2	132.1	166.2	106.3
mov(w)	-2.4	-4.1	3.2	-6.6	0.6	-3.4	-3.7	-2.3
load/store	75.3	80.6	78.7	31.8	36.3	56.7	67.9	61.0
other	56.5	45.1	66.0	22.7	36.6	18.7	104.6	50.0
CODE SIZE OVERHEAD REDUCTION PER OPTIMISATION (%)								
Impr. peephole	-43.6	-45.1	-51.1	-25.4	-36.6	-52.5	-66.5	-45.8
Stack caching	-9.4	-35.5	-26.6	-30.6	-71.6	-79.7	-100.9	-50.6
Pop. val. caching	-37.6	-39.5	-10.6	-17.1	-7.6	-19.1	-20.7	-21.8
Mark loops	-14.1	-12.0	-10.6	-6.1	+ 10.2	+ 0.2	-10.1	-6.0
Const shift	0.0	-3.7	-5.4	-5.0	-0.6	+ 0.8	-2.6	-2.4
CODE SIZE OVERHEAD AFTER OPTIMISATIONS (%)								
Total	107.1	73.9	145.7	22.7	81.5	53.8	134.1	88.4
push/pop	25.9	6.0	25.5	4.3	17.4	0.4	7.0	12.4
mov(w)	5.9	4.1	9.6	-2.6	5.6	-2.1	17.8	5.5
load/store	21.2	25.4	53.2	3.8	19.8	36.8	23.5	26.2
other	54.1	38.4	57.4	17.3	38.5	18.7	85.7	44.3

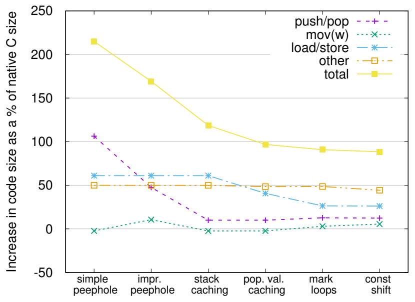

Figure 4. Code size overhead per category

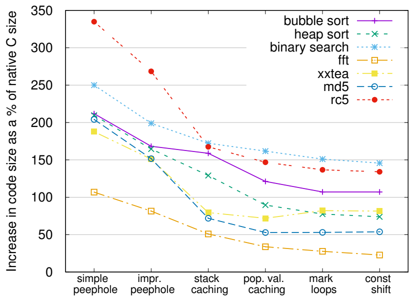

Figure 5. Code size overhead per benchmark

compare different benchmarks, we normalise this overhead to a percentage of the number of bytes or cpu cycles used by the native implementation: a 100% overhead means the AOT compiled version takes twice as long to run, or twice as many bytes to store. The exact results can vary depending on factors such as which benchmarks are chosen, the input data, etc., but the general trends are all quite stable.

Using the trace data produced by Avrora, we get a detailed view into the runtime performance and the different types of overhead. We count the number of bytes and cycles spent on each native instruction for both the native C and our AOT compiled version, and then group them into 4 categories that roughly match the 3 types of overhead:

- PUSH,POP: Matches the type 1 push/pop overhead since native code uses almost no push/pop instructions.
- LD,LDD,ST,STD: Matches the type 2 load/store overhead and directly shows the amount of memory traffic.
- MOV,MOVW: For moves the picture is less clear since the AOT compiler emits them for various reasons. Before we introduce stack caching, it emits moves to replace push/pop pairs, and after the mark loops to save a pinned value when it is popped destructively.
- others: the total overhead, minus the previous three categories. This roughly matches the type 3 overhead.

We define the overhead from each category as the number of bytes or cycles spent in the AOT version, minus the number spent by the native version, and again normalise this to the *total* number of bytes or cycles spent in the native C version. The detailed results for each benchmark and type of overhead are shown in tables 5 and 6.

6.1 Performance

In Figure 2 we see how our optimisations combine to reduce performance overhead. We take the average of the 7 benchmarks, and show both the total overhead, and the overhead for each instruction category. Figure 3 shows the total overhead for each individual benchmark. We start with the original version with only the simple peephole optimiser, and then incrementally add our five optimisations.

Using the simple optimiser, the type 1 and type 2 overhead are similar, at 148% and 115%, while the type 3 overhead is less significant at 67%. The basic approach does not have many reasons to emit a move, so we see the AOT version actually spends fewer cycles on move instructions than the C version, resulting in a small negative value. When we improve the peephole optimiser to include non-consecutive push/pop pairs, push/pop overhead drops by 97%. But if the push and pop target different registers, they are replaced by a move instruction, and we see an increase of 11% in move overhead. For a 16-bit value this takes 1 cycle (for a MOVW instruction), instead of 8 cycles for two pushes and two pops. The increase in moves shows most of the extra cases that are handled by the improved optimiser are replaced by a move instead of eliminated, since the 11% extra move overhead matches an 88% reduction in push/pop overhead.

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

Having eliminated the type 1 overhead almost completely, we now add popped value caching to remove a large number of the unnecessary load instructions. This reduces the mem-

Table 7. Code size and memory consumption

	size vs interpreter	AOT code reduction	break even	memory usage
simple peephole	4760B			20B
improved peephole	5024B	-14.5%	1.8KB	20B
stack caching	6122B	-30.6%	4.5KB	31B
pop. val. caching	7306B	-37.5%	6.8KB	79B
mark loops	10164B	-39.4%	13.7KB	86B
const shift	10758B	-40.2%	14.9KB	87B

ory traffic significantly, as is clear from the reduced load/store overhead, while the other types remain stable. Adding the mark loops optimisation further reduces loads, and this time also stores, by pinning common variables to a register, but it uses slightly more move instructions, and the fact that we have fewer registers available for stack caching means we have to spill stack values to memory more often. While we save 53% on loads and stores, the push/pop and move overhead both increase by 7%.

Both the push/pop and load/store overhead have now been almost 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 we can eliminate over one third of it by optimising constant shifts.

Combined, these optimisations reduce performance overhead from 327% to 68% of native C performance.

6.2 Code size

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

The size overhead for the generated code is shown in figures 4 and 5, again split up per instruction category and benchmark respectively. For the first three optimisations, the two graphs follow a similar pattern as the performance graphs. These optimisations eliminate the need to emit certain instructions, which reduces code size and improves performance at the same time. For the mark loops and constant shift optimisations, the effect on code size is much smaller.

The mark loops optimisation moves loads and stores for pinned variables outside of the loop. This results in a 6% reduction of code size overhead, much less than the 39% reduction in performance overhead. This has two reasons: first, for stores and loads that are popped destructively, we still need to emit a move, which saves 3 cycles, but only 1 instruction. A more important reason however, is that we get the performance advantage at each run time iteration, but the code size advantage of emitting less instructions only once.

The constant shift optimisation unrolls the loop that is normally generated for bit shifts. This significantly improves performance, but the effect on the code size depends on the number of bits to shift by. The constant load and loop take at least 5 instructions. In most cases the unrolled shifts will be smaller, but md5 actually shows a small 1% increase in code size since it contains many shifts by a large number of bits.

6.2.1 VM code size and break-even points

Obviously, more complex code generation techniques will increase the size of our compiler. The first column in Table 7 shows the difference in code size between the AOT translator and Darjeeling's interpreter. The basic AOT approach is 4760B larger than the interpreter, and our optimisations each add a little to the size of the VM.

They also generate significantly smaller code. The second column shows the reduction in the generated code size compared to the baseline approach. Here we show the reduction in total size, as opposed to the overhead used elsewhere, to be able to calculate the break-even point. Using the improved peephole optimiser adds 264 bytes to the VM, but it reduces the size of the generated code by 14.5%. If we have more than 1.8KB available to store user programmes, this reduction will outweigh the increase in VM size. More complex optimisations further increase the VM size, but compared to the baseline approach, the break-even point is well within the range of memory typically available, peaking at 15KB when all optimisations are included.

As is often the case, there is a tradeoff between size and performance. The interpreter is smaller than each version of our AOT compiler, and Table 6 shows JVM bytecode is smaller than both native C and AOT compiled code, but the interpreter's performance penalty may be unacceptable in many cases. Using AOT compilation we can achieve adequate performance, but the most important drawback has been an increase in generated code size. These optimisations help to significantly mitigate this drawback, and both improve performance, and allow us to load more code on a device.

For the smallest devices we may decide to use only the first three optimisations to limit the VM size and still get both a reasonable performance, and most of the code size reduction. This reduces code size by 37.5%, and results in a performance overhead of 136%.

6.2.2 VM memory consumption

The last column in Table 7 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 20 bytes on keeping a number of commonly used values such as a pointer to the next instruction to be compiled, the number of instructions in the method, etc. The simple stack caching approach adds a 11 byte array to store the state of each register pair we use for stack caching. Popped value caching adds two more arrays of 16-bit elements to store the value tag and age of each value. Mark loops only needs an extra 16-bit word to mark which registers are pinned, and a few other variables. Finally, the constant shift optimisation only requires a single byte. In total, our compiler requires 87 bytes of memory during the compilation process.

6.3 Benchmark details

Finally 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 5 shows the distribution of the JVM instructions executed in each benchmark, and both the maximum and average number of bytes on the JVM stack. We can see some important differences between the benchmarks. While the left most are almost completely load/store bounded, towards the right the benchmarks become more computation intensive, spending fewer instructions on loads and stores, and more on math or bitwise operations. The left benchmarks have only a few

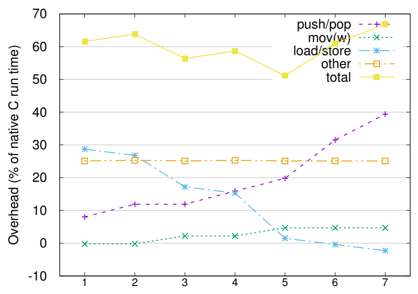

Figure 6. Xxtea performance overhead for different number of pinned register pairs

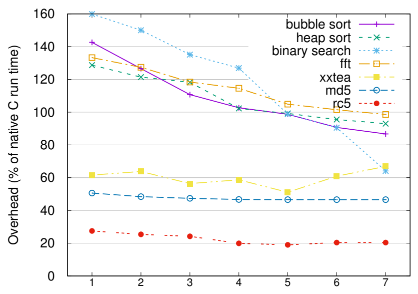

Figure 7. Per benchmark performance overhead different number of pinned register pairs

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

The second part of tables 5 and 6 first shows the overhead before optimisation, split up in the four instruction categories. We then list the effect of each optimisation on the total overhead. Finally we show the overhead per category after applying all optimisations.

The improved peephole optimiser and stack caching both target the push/pop overhead. Stack caching can eliminate almost all, and replaces the need for a peephole optimiser, but it is interesting to compare the two. The improved peephole optimiser does well for the simple benchmarks like sort and search, leaving less overhead to remove for stack caching. Moving to the right, the more complicated expressions mean there is more distance between a push and a pop, leaving more cases that cannot be handled by the peephole optimiser, and replacing it with stack caching yields a big improvement.

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

Bit shifts Interestingly, the reason fft is the slowest, is similar to the reason rc5 is fastest: they both spend a large amount of time doing bit shifts. Rc5 shifts by a variable, but large number of bits. Only 7.7% of the executed JVM instructions are bit shifts, but they account for 52% of the execution time. For these variable bit shifts, our translator and avr-gcc generate a similar loop, so the two share a large constant factor.

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

Bubble sort Next we look at bubble sort in some more detail. After optimisation, we see most of the stack related overhead has been eliminated and of the 87% 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 87% overhead, 72% is caused by these address calculations.

Xxtea and the mark loops optimisation Perhaps the most interesting benchmark is xxtea. Its high average stack depth means popped value caching does not have much effect: most registers are used for real stack values, leaving few chances to reuse a value that was previously popped from the stack.

When we apply the mark loops optimisation, performance actually degrades by 11.7%, and code size overhead increases 10.2%! 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 7 register pairs to pin variables to was also the best option. At a lower average stack depth, the fewer number of registers available for stack caching is easily compensated for by the cheaper variable access. For xxtea however, the cost of spilling more stack values to memory outweighs the gains of pinning more variables when too many variables are pinned. Figure 6 shows the overhead for xxtea from the different instruction categories. When we increase the number of register pairs used to pin variables from 1 to 7, the load/store overhead steadily decreases, but the push/pop and move overhead increase. The optimum is at 5 pinned register pairs, at which the total overhead is only 51%, instead of 67% at 7 pinned register pairs.

Figure 7 shows the performance for each benchmark, as the number of pinned register pairs is increased. The three benchmarks that stay stable or even slow down when the number pinned pairs is increased beyond 5 are exactly the benchmarks that have a high stack depth: xxtea, md5 and rc5. It should be possible to develop a simple heuristic to allow the VM to make a better decision on the number of registers to pin. Since our current VM always pins 7 pairs, we used this as our end result and leave this heuristic to future work.

7 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] 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.

In this paper, we presented techniques to reduce the code size overhead after AOT compilation by 59% and the performance overhead by 79%, resulting in a compiler that produces code that is on average only 1.7 times slower and 1.9 times larger than optimised C.

Our optimisations do increase the size of our VM, but the break-even point at which this is compensated for by the smaller code it generates, is well within the range of programme memory typically available on a sensor node. This leads us to believe that these optimisations will be useful in many scenarios, and make using a VM a viable option for a wider range of applications.

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

A more general question is what the most suitable architecture and instruction set is for a VM on tiny devices. Hsieh et al. note that the performance problem lies in the mismatch between the VM and the native machine architecture [16]. In fact, we believe JVM is probably 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 [17] 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 [29, 25]. 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.

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

8 References

[1] Connected Limited Device Configuration. `http://www.oracle.com/technetwork/java/cldc-141990.html`.

[2] LibTomCrypt. `http://www.libtom.org/LibTomCrypt`.

[3] B. Alpern et al. Implementing Jalapeño in Java. In *OOPSLA*, 1999.

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

[5] F. Aslam et al. Introducing TakaTuka - A Java Virtualmachine for Motes. In *SENSYS*, 2008.

[6] R. Balani et al. Multi-level Software Reconfiguration for Sensor Networks. In *EMSOFT*, 2006.

[7] N. Brouwers, K. Langendoen, and P. Corke. Darjeeling, A Feature-rich VM for the Resource Poor. In *SENSYS*, 2009.

[8] A. Courbot, G. Grimaud, and J.-J. Vandewalle. Efficient Off-board Deployment and Customization of Virtual Machine-based Embedded Systems. *ACM Trans. Embed. Comput. Syst.*, 9(3), 2 2010.

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

[10] J. Ellul and K. Martinez. Run-Time Compilation of Bytecode in Sensor Networks. In *SENSORCOMM*, 2010.

[11] M. A. Ertl. Stack Caching for Interpreters. In *PLDI*, 1995.

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

[13] A. Gal, C. W. Probst, and M. Franz. HotpathVM: An Effective JIT Compiler for Resource-constrained Devices. In *VEE*, 2006.

[14] T. Harbaum. NanoVM. `http://harbaum.org/till/nanovm/index.shtml`.

[15] K. Hong et al. TinyVM, an Efficient Virtual Machine Infrastructure for Sensor Networks. In *SENSYS*, 2009.

[16] C.-H. A. Hsieh, J. C. Gyllenhaal, and W.-m. W. Hwu. Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results. In *MICRO*, 1996.

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

[18] J. Koshy and R. Pandey. VM*: Synthesizing Scalable Runtime Environments for Sensor Networks. In *SENSYS*, 2005.

[19] A. Krall. Efficient JavaVM Just-in-Time Compilation. In *PACT*, 1998.

[20] P. Levis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *ASPLOS*, 2002.

[21] P. Levis, D. Gay, and D. Culler. Active Sensor Networks. In *NSDI*, 2005.

[22] K. Lorincz et al. Mercury: A Wearable Sensor Network Platform for High-Fidelity Motion Analysis. In *SENSYS*, 2009.

[23] R. Müller, G. Alonso, and D. Kossmann. A Virtual Machine for Sensor Networks. In *EuroSys*, 2007.

[24] G. Muller et al. Harissa: A Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code. In *COOTS*, 1997.

[25] Y. Shi et al. Virtual Machine Showdown: Stack Versus Registers. In *VEE*, 2005.

[26] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: Scalable Sensor Network Simulation with Precise Timing. In *IPSN*, 2005.

[27] D. Wheeler and R. Needham. XXTEA: Correction to XTEA. Technical report, Computer Laboratory, University of Cambridge, 1998.

[28] I. Wirjawan et al. Balancing Computation and Communication Costs: The Case for Hybrid Execution in Sensor Networks. *Elsevier Ad Hoc Networks*, 6(8), Nov. 2008.

[29] Y. Zhang et al. Swift: A Register-based JIT Compiler for Embedded JVMs. *ACM SIGPLAN Notices*, 47(7), Jan. 2012.