國立臺灣大學電機資訊學院資訊工程學系
博士論文
Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Taiwan University
Doctoral Dissertation

我的 VM
My VM

賴爾思
Niels Reijers

指導教授：施吉🄵教授
Advisor: Professor Chi-Sheng Shih

中華民國 107 年 3 月
March, 2018

國立臺灣大學博士學位論文
口試委員會審定書

我的 VM
My VM

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

口試委員:

_____

_____ _____

_____ _____

_____ _____

_____ _____

所　長: _____

Acknowledgements

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

摘要

中文摘要

關鍵字： 關鍵字

Abstract

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

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

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

Using a set of 12 benchmarks with varying characteristic, including the standard CoreMark benchmark, and a number of real sensor node applications, we show this results in an average performance roughly 2.1x slower than unsafe native code. Without safety checks, this drops to 1.7x. Thus,

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

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

Contents

List of Figures

List of Tables

List of Listings

Table 1: Code size data per benchmark

	B.sort	H.sort	BinSearch	XXTEA	MD5	RC5	FFT	Outlier	LEC	CoreMark	MoteTrack	HeatCalib	HeatDetect	average
CODE SIZE (BYTES)														
JVM	74	136	83	379	2983	453	445	287	338	2656	2329	311	2733	
Native C	118	298	146	1442	9458	910	1292	380	560	6128	3906	1944	5294	
AOT original	414	988	408	3748	29330	4054	2582	1394	1660	13562	11422	2448	17886	
AOT optimised	258	596	310	2236	14654	2018	1342	800	1074	9182	7852	1610	10994	
CODE SIZE OVERHEAD BEFORE OPTIMISATIONS (% of nat. C)														
Total	250.8	231.5	179.5	159.9	210.1	345.5	99.8	266.8	196.4	121.3	192.4	25.9	237.9	193.7
push/pop	71.2	85.9	60.3	102.6	133.1	164.8	53.9	86.3	68.6	57.0	69.5	34.8	95.1	83.3
load/store	88.1	79.2	74.0	28.4	56.7	67.9	20.9	101.1	75.4	44.7	59.8	23.5	56.4	59.7
mov(w)	6.8	4.0	1.4	0.7	-2.7	-1.3	2.9	2.6	5.0	-4.8	12.7	-15.6	13.5	1.9
other	84.7	62.4	43.8	28.2	22.9	114.1	22.1	76.8	47.5	24.4	50.4	-16.7	72.8	48.7
CODE SIZE OVERHEAD REDUCTION PER OPTIMISATION (% of nat. C)														
Impr. peephole	-67.7	-55.0	-45.3	-38.5	-54.3	-64.2	-33.2	-75.7	-37.1	-24.7	-21.7	-14.4	-49.0	-44.7
Stack caching	-22.1	-22.8	-21.9	-56.1	-78.7	-105.5	-18.6	-30.6	-38.2	-24.9	-24.2	-14.2	-43.8	-38.6
Pop. val. caching	-16.9	-37.6	-6.8	-6.2	-18.7	-18.7	-13.8	-5.2	-19.3	-11.6	-24.1	-8.1	-20.0	-15.9
Mark loops	+1.7	+10.1	+21.9	+5.9	-1.2	-2.6	-4.2	-16.4	+2.5	+1.7	-7.4	-2.5	-6.0	0.3
Const shift	0.0	-3.4	-6.9	+1.7	+2.8	-16.0	-4.6	-2.6	-1.8	-1.1	0.0	-1.7	-0.7	-2.7
16-bit array index	-27.2	-22.8	-8.2	-11.6	-5.1	-16.7	-11.6	-25.8	-10.7	-7.4	-14.0	-2.2	-10.7	-13.4
SIMUL	0.0	0.0	0.0	0.0	0.0	0.0	-9.9	0.0	0.0	-3.5	0.0	0.0	0.0	-1.0
CODE SIZE OVERHEAD AFTER OPTIMISATIONS (% of nat. C)														
Total	118.6	100.0	112.3	55.1	54.9	121.8	3.9	110.5	91.8	49.8	101.0	-17.2	107.7	77.7
push/pop	23.7	16.1	27.4	13.3	0.0	6.2	2.5	-2.1	-3.6	4.8	16.3	3.9	0.2	8.4
load/store	33.9	41.6	49.3	14.8	37.2	25.3	-1.7	57.9	46.8	28.2	36.6	8.0	37.2	31.9
mov(w)	1.7	6.7	6.8	2.5	-2.4	11.9	-0.3	1.1	8.2	-0.6	13.2	-10.7	15.0	4.1
other	59.3	35.6	28.8	24.4	20.1	78.5	3.4	53.7	40.4	17.5	35.0	-18.4	55.2	33.3
	B.sort	H.sort	BinSearch	XXTEA	MD5	RC5	FFT	Outlier	LEC	CoreMark	MoteTrack	HeatCalib	HeatDetect	average

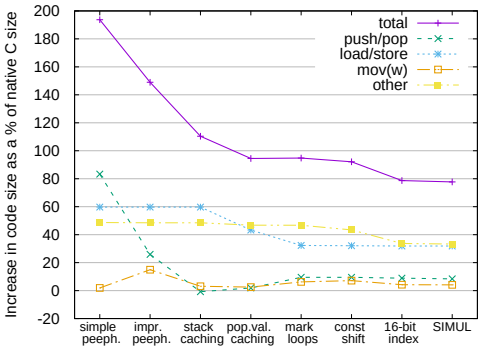


Figure 1: Code size overhead per category

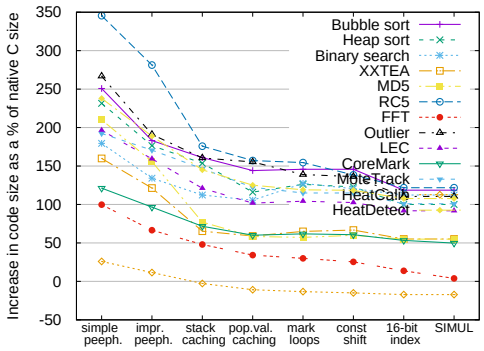


Figure 2: Code size overhead per benchmark

0.1 AOT translation: code size

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

The size overhead for the generated code is shown in figures 1 and 2, again split up per instruction category and benchmark respectively. For the first three optimisations, the two graphs follow a similar pattern as the performance graphs. These optimisations eliminate the need to emit certain instructions, which reduces code size and improves performance at the same time.

The mark loops optimisation moves loads and stores for pinned variables outside of the loop. This reduces performance overhead by 41%, but the effect on code size varies per benchmark: some are slightly smaller, others slightly larger.

For each value that is live at the beginning of the loop, we need to emit the load before the mark loop block, so in terms of code size we only benefit if it is loaded more than once, and may actually lose some if it is then popped destructively, since we would need to emit a `mov`. Stores follow a similar argument. Also, for small methods the extra registers used may mean we have to save more call-saved registers in the method prologue. Finally, we get the performance advantage for each run-time iteration, but the effect on code size, whether positive or negative, only once.

The constant shift optimisation unrolls the loop that is normally generated for bit shifts. This significantly improves performance, but the effect on the code size depends on the

number of bits to shift by. The constant load and loop take at least 5 instructions. In most cases the unrolled shifts will be smaller, but MD5 and XXTEA show a small increase in code size since they contains shifts by a large number of bits.

Using 16-bit array indexes also reduces code size. The benchmarks here already have the manual code optimisations, so they use short index variables. This means the infuser will emit a `S2I` instruction to cast them to 32-bit ints if the array access instructions expect an int index. Not having to emit those when the array access instructions expect a 16-bit index, and the reduced work the access instruction needs to do, saves about 13% code size overhead in addition to the 17% reduction in performance overhead. Using 32-bit variables in the source code also removes the need for `S2I` instructions, but the extra effort needed to manipulate the 32-bit index variable makes the net code size even larger.

0.1.1 VM code size and break-even point

These more complex code generation techniques do increase the size of our compiler. The first column in Table 2 shows the difference in code size between the AOT translator and Darjeeling's interpreter. The basic AOT approach is 6605B larger than the interpreter, and each of our optimisations adds a little to the size of the VM.

They also generate significantly smaller code. The third column shows the reduction in the generated code size compared to the baseline approach. Here we show the reduction in total size, as opposed to the overhead used elsewhere, to be able to calculate the break-even point. Using the improved peephole optimiser adds 276 bytes to the VM, but it reduces the size of the generated code by 14.8%. If we have more than 1.9KB available to store user programmes, this reduction will outweigh the increase in VM size. Adding more complex optimisations further increases the VM size, but compared to the baseline approach, the break-even point is well within the range of memory typically available on a sensor node, peaking at at most 18.3KB.

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 1 shows JVM bytecode is smaller than both native C and AOT compiled code, but the interpreter's performance

Table 2: Code size and memory consumption

	size vs interpreter	size vs baseline		AOT code reduction	break even	memory usage
Baseline	6605 B					23 B
Improved peephole	6881 B	276 B	(+276)	-14.8%	1.9 KB	23 B
Simple stack caching	7825 B	1220 B	(+944)	-27.6%	4.4 KB	34 B
Popped value caching	9057 B	2452 B	(+1232)	-33.1%	7.4 KB	78 B
Markloop	12337 B	5732 B	(+3280)	-33.0%	17.4 KB	85 B
Const shift	12785 B	6180 B	(+448)	-33.8%	18.3 KB	85 B
16-bit array index	12765 B	6160 B	(-20)	-38.2%	16.1 KB	85 B
SIMUL	12831 B	6226 B	(+66)	-38.7%	16.1 KB	85 B
Lightweight methods	13379 B	6774 B	(+548)	-38.9%	17.4 KB	85 B

penalty may be unacceptable in many cases. Using AOT compilation we can achieve adequate performance, but the most important drawback has been an increase in generated code size. These optimisations help to mitigate this drawback, and both improve performance, and allow us to load more code on a device.

For the smallest devices, or if we want to be able to load especially large programmes, we may decide to use only a selection of optimisations to limit the VM size and still get both a reasonable performance, and most of the code size reduction. For example, dropping the markloop and constant shift optimisations would reduce the size of the VM by 3.5KB while maintaining most of the reduction in generated code size, while performance overhead would increase to about 126%.

0.1.2 VM memory consumption

The last column in Table 2 shows the amount of data that needs to be kept in memory while translating a method. We would like our VM to be able to load new code while other tasks are running concurrently. Here we only list the data that the VM would need to maintain in between receiving messages with new code over the air since this indicates the amount of memory that needs to be reserved and would not be available to other tasks during this process. Of course while a message with new code is being processed, more stack memory is used, but this is freed again after a message has been processed and can be reused by other applications.

For the baseline approach we only use 23 bytes for a number of commonly used values

such as a pointer to the next instruction to be compiled, the number of instructions in the method, etc. The simple stack caching approach adds a 11 byte array to store the state of each register pair we use for stack caching. Popped value caching adds two more arrays of 16-bit elements to store the value tag and age of each value. Mark loops only needs an extra 16-bit word to mark which registers are pinned, and a few other variables. Finally, the instruction set optimisations do not require any additional memory. In total, our compiler requires 85 bytes of memory during the compilation process.

0.2 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 **??** 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 sort benchmarks on the left are almost completely load/store bounded, XXTEA, RC5 and MD5 are much more computation intensive, spending fewer instructions on loads and stores, and more on math or bitwise operations. The left three benchmarks and the outlier detection benchmark have only a few bytes on the stack, but as the benchmarks contain more complex expressions, the number of values on the stack increases.

The second part of tables **??** and 1 first shows the overhead before optimisation, split up in the five instruction categories. We then list the effect of each optimisation on the total overhead. Finally we show the overhead per category after applying all optimisations.

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

marks, replacing the peephole optimiser with stack caching yields a big improvement.

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

The smaller benchmarks highlight certain specific aspects of our approach, while the larger CoreMark benchmark covers a mix of different types of processing. As a result, it is an average case in almost every row in Table **??**. The reason it ends up being the third slowest after all optimisations was discussed in **??**. With the 'unfair' optimisations described there, CoreMark's performance overhead would be 61%, very close to the average of the other benchmarks.

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 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, 79.8% is due to `ADD` instructions, but bubble sort hardly does any additions. This is a good example of how the simple JVM instruction set leads to less efficient code. To access an array we need to calculate the address of the indexed value, which takes one move and five additions for an array of shorts. This calculation is repeated for each access, while the C version 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 Looking at the code size data in Table 1, we see the HeatCalib benchmark has a negative code size overhead. This is caused by the fact that we compile the C versions using `avr-gcc`'s -O3 optimisations, optimising for performance instead of code size. In this case, as well as for FFT, this caused `avr-gcc` to duplicate a part of 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 165% overhead compared to native C. MoteTrack stores a database of reference signatures in Flash. In C this is a complex struct containing a number of sub-structures and fixed-sized arrays, while it becomes a collection of objects and arrays in Java, shown in 1.1.

Since the layout of the complete C structure is known at compile time, the C function to load a reference signature from its database can simply use `memcpy_P` to copy a block of 80 bytes from Flash to RAM. For Java, the method to read from Flash must follow multiple pointers to follow several indirections to find the locations to put each value. As a result, reading a single signature takes 1695 cycles in Java, and only 735 cycles in C.

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

A compression algorithm like LEC is a good example of an optimisation that may be part of an application loaded onto a sensor node. However, if the overhead of using a VM is too high, the cost of compression may outweight the energy saved on transmission. Table **??** shows that using the baseline AOT approach, the LEC benchmark has an overhead of 278.4%. Because the CPU has to stay active longer, this means compressing the data would cost $246\mu J * 3.784 = 930\mu J$, which is more than the 650 µJ saved on transmission.

After we apply our optimisations, the overhead is reduced to 86.5%, resulting in $246\mu J * 1.685 = 459\mu J$ spent on compression. While the savings are less than when we use native C to compress the data, our optimisations mean we can save on transmission costs by using LEC compression, while using the baseline AOT approach, LEC compression would have resulted in a net loss.

Xxtea and the mark loops optimisation 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.

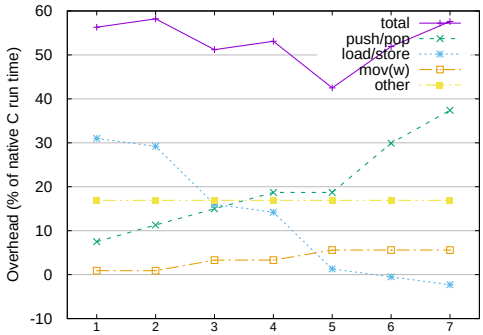


Figure 3: Xxtea performance overhead for different number of pinned register pairs

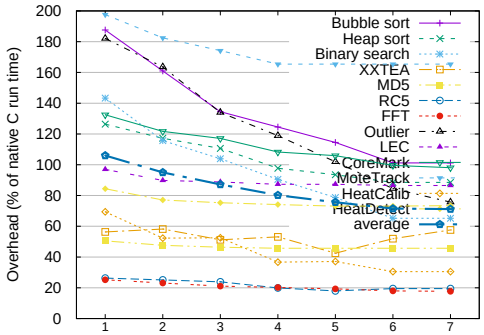


Figure 4: Per benchmark performance overhead different number of pinned register pairs

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 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 3 shows the overhead for xxtea from the different instruction categories. When we increase the number of register pairs used to pin variables from 1 to 7, the load/store overhead steadily decreases, but the push/pop and move overhead increase. For XXTEA, the optimum is at 5 pinned register pairs, at which the total overhead is only 43%, instead of 58% at 7 pinned register pairs.

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

Figure 4 shows the performance for each benchmark, as the number of pinned regis-

Table 3: Methods per benchmark and relative performance for normal, lightweight invocation, and inlining. Highlights indicate changes from the versions used to obtain the results in the previous sections.

	# calls	C	Java Base version	Java Alternative version	Java Using normal method calls
CoreMark					
ee_isdigit	3920	normal (inlined)	manually inlined	lightweight (JVM)	manually inlined
core_state_transition	1024	normal	lightweight	lightweight	normal
crcu8	584	normal (inlined)	lightweight	lightweight	normal
crcu16	292	normal	lightweight	lightweight	normal
calc_func	220	normal	lightweight	lightweight	normal
compare_idx	209	normal (inlined)	normal (virtual)	normal (virtual)	normal (virtual)
core_list_find	206	normal	lightweight	lightweight	normal
compare_complex	110	normal	normal (virtual)	normal (virtual)	normal (virtual)
crcu32	64	normal	lightweight	lightweight	normal
matrix_sum	16	normal	lightweight	lightweight	normal
others (<16 calls each)	39	normal	normal	normal	normal
cycles			3,373,037	3,543,531	4,844,211
overhead v native C			98.1%	108.1%	184.5%
code size			9,182	9,228	9,512
FFT					
FIX_MPY	768	marked inline	manually inlined	lightweight(JVM)	normal
cycles			179,818	215,146	661,486
overhead v native C			17.7%	40.9%	333.1%
code size			1,342	1,338	1,426
heap sort					
SWAP	1642	#define	manually inlined	manually inlined	manually inlined
siftDown	383	normal	lightweight	manually inlined	normal
cycles			208,239	184,071	437,264
overhead v native C			88.5%	66.6%	295.8%
code size			596	662	602

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

0.3 Method invocation

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

Most of our smaller benchmarks consist of only a single method. ProGuard automatically inlines methods only called from a single location, eliminating all method calls in the *LEC* benchmark. We will examine the effect of lightweight methods using the *FFT*, *Heap sort*, and *CoreMark* benchmarks. FFT contains a single function, `fix_mpy`, which was inlined by the C compiler, so we manually inlined it in the Java version. Heap sort contains a real method call: it consist of two loops, both repeatedly calling the `siftDown` method, so ProGuard doesn't inline it. Since it is called from two places, and larger than ProGuard's size threshold, it is not automatically inlined. CoreMark is a much more extensive benchmark, and consists of many methods.

In Table 3 we see the most frequently called methods of the CoreMark, the two methods in FFT and heap sort benchmarks, and the number of times they are called in a single run. Next, we list the way they are implemented in C. CoreMark only defines normal functions, which are inlined by `avr-gcc` in three cases. FFT's `fix_mpy` function is marked with the `inline` compiler hint, which was followed by `avr-gcc`. Finally heap sort uses just one extra function, and a macro to swap two array elements.

The Java base version column shows the way these functions are implemented in the Java versions of our benchmarks. We manually inlined C macros, and the functions that were inlined by the C compiler. The most commonly called methods were transformed to lightweight methods, simply by adding the `@Lightweight` annotation where possible. For Java versions of the `compare_idx` and `compare_complex` methods, this was not possible since we do not support lightweight virtual methods.

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

For the CoreMark benchmark, we first replace the inlined implementation of the most frequently called method with a lightweight version. `ee_isdigit` returns true if a `char` passed to it is between `'0'` and `'9'`. Since this is a very trivial method, we manually coded the lightweight method to use only the stack and no local variables. This slowed down the benchmark by 5%, adding 170,494 cycles. Since the method is called 3920 times, this corresponds to an overhead of about 44 cycles, which is on the high side for

such a small method.

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

Next, we see what the performance would be without lightweight methods, and all methods, except the manually inlined `ee_isdigit`, have to be implemented as normal Java methods. This adds a total of 1,471,174 cycles, making it almost 1.45 times slower than the lightweight methods version. Spread over 2,725 calls, this means the average method invocation added over 540 cycles, which is within the range predicted in Section **??**.

The FFT benchmark has a much lower running time than CoreMark, but still does 768 function calls. In the C and normal Java versions these are inlined. When we change `fix_mpy` to a normal Java method, it is too large for ProGuard to inline. If we mark it `@Lightweight` the large number of calls relative to the total running time means the average overhead of over 41 cycles per invocation slows down the benchmark by 18%. Without lightweight methods, this would be as high as 268%

Finally, for the heap sort benchmark we normally use a lightweight method for `siftDown`. While manually inlining it is possible, it is not an attractive option since the `siftDown` method is much more complex than `FIX_MPY` or `ee_isdigit`. When we do inline it, we see that the lightweight method adds some overhead compared to the inlined version, but much less than a normal method call would.

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

The difference between inlining and lightweight methods is less clear. For the smallest of methods, such as CoreMark's `ee_isdigit`, the inlined code is smaller than the call, but the heap sort benchmark shows that inlining larger methods can result in significantly

larger code. Since `siftDown` is called from two places, duplicating it leads to a 11% increase for the 16-bit version of heap sort. For the 32-bit version, where `siftDown` is relatively larger, this increases to 27%.

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

0.4 The cost of safety

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

,

Table 4: Cost of safety guarantees

	B.sort	H.sort	BinSearch	XXTEA	MD5	RC5	FFT	Outlier	LEC	CoreMark	MoteTrack	HeatCalib	HeatDetect	average
EXECUTED JVM INSTRUCTIONS (% of total executed JVM instructions)														
Array element/object field STORES	18.0	7.8	0.0	2.9	4.5	1.5	6.1	5.8	3.6	2.6	6.6	1.4	4.7	5.0
Array element/object field LOADS	18.0	15.9	7.1	8.6	6.2	6.4	6.9	10.7	7.9	11.7	18.9	4.1	9.8	10.2
PERFORMANCE OVERHEAD VS NATIVE C (% of nat. C)														
unsafe	101.2	88.5	65.2	57.6	45.7	19.5	17.7	75.7	86.5	98.1	165.4	30.5	73.4	71.2
safe writes	247.5	153.9	65.2	68.2	60.3	22.2	30.4	128.4	120.2	125.2	222.6	33.9	91.3	105.3
safe reads and writes	393.9	287.8	151.7	100.0	80.3	33.4	44.9	226.6	193.2	203.4	382.6	43.9	126.9	174.5
PERFORMANCE OVERHEAD VS UNSAFE VM (% of unsafe AOT)														
safe writes	72.7	34.7	0.0	6.7	10.0	2.3	10.8	30.0	18.1	13.7	21.6	2.6	10.3	19.9
safe reads and writes	145.5	105.7	52.4	26.9	23.7	11.6	23.1	85.9	57.2	53.2	81.8	10.3	30.9	60.3
CODE SIZE OVERHEAD VS NATIVE C (% of nat. C)														
unsafe	118.6	100.0	112.3	55.1	54.9	121.8	3.9	110.5	91.8	49.8	101.0	-17.2	107.7	77.7
safe writes	125.4	105.4	112.3	56.2	55.7	125.3	6.3	118.9	97.5	53.7	107.9	-16.4	114.7	81.8
safe reads and writes	132.2	113.4	117.8	60.1	59.1	132.3	10.1	123.2	107.5	61.0	127.0	-13.9	118.5	88.3
CODE SIZE OVERHEAD VS UNSAFE VM (% of unsafe AOT)														
safe writes	3.1	2.7	0.0	0.7	0.5	1.6	2.3	4.0	3.0	2.6	3.4	1.0	3.4	2.3
safe reads and writes	6.2	6.7	2.6	3.2	2.7	4.7	6.0	6.0	8.2	7.5	12.9	4.0	5.2	6.0

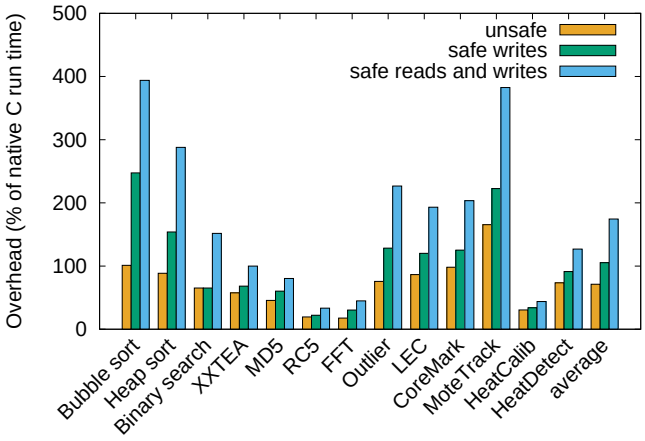


Figure 5: Overhead increase due to safety checks

0.4.1 Run-time cost

First we examine the run-time cost of our safety guarantees. Table 4 shows the results for our 12 benchmarks. Figure 5 shows the data in the first section of the table. The baseline here is the unsafe version of our VM, which is on average 71.1% slower than native C. We first add write checks only, since this is sufficient to satisfy our guarantee that no malicious code can corrupt the state of the VM. This increases the average overhead to 105.3% of native C, corresponding to an 19.9% increase in runtime compared to the unsafe VM.

We can see the cost of safety depends greatly on the benchmark we run. Most checks are done at load-time, including writes to local and static variables. The only check that adds significant run-time overhead is check **??**, which checks the target of an object field or array write is within the heap bounds.

Thus, the run-time overhead is determined by the number of object or array accesses a benchmark does. The percentage of these is shown in the first part of Table 4. Since bubble sort has by far the highest percentage of array writes, at 18% of all executed bytecode instructions, it also incurs the highest overhead from adding write safety, slowing down by 47%. Binary search on the other hand, which does no writes at all, is unaffected. As usual CoreMark, being a large benchmark with a mix of operations, is somewhere in the middle. The clear correlation between the percentage of array and object writes, and the slowdown compared to the unsafe version is shown in Figure 6.

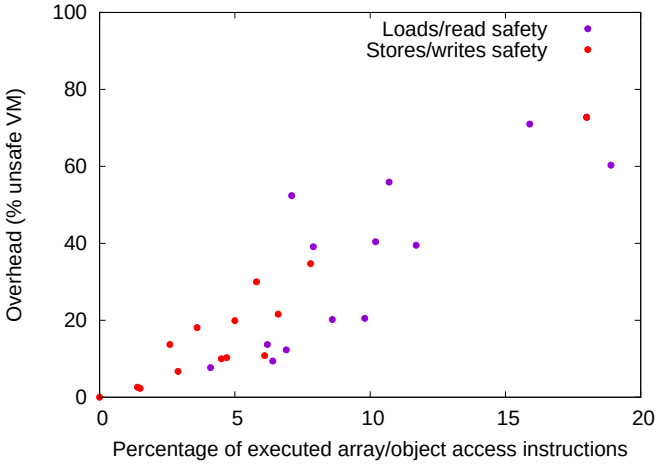


Figure 6: Percentage of array/object load/store instructions and cost of read/write safety

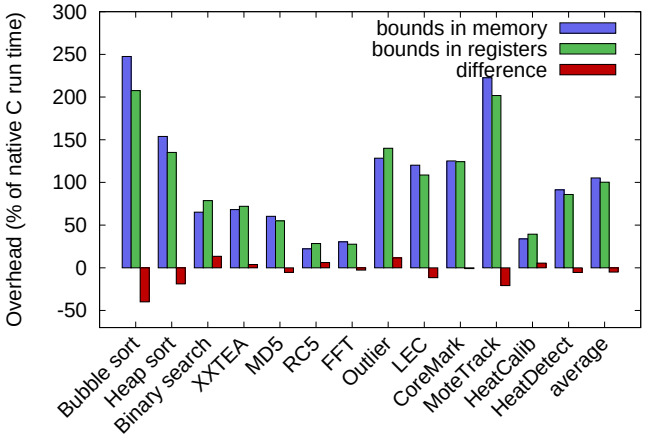


Figure 7: Comparison of safety cost with bounds in memory or registers

Safe reads Next we consider read safety. Up to this point our VM only checks the application cannot *write* to memory it's not supposed to write to, however, it may still read from any location.

The recently published Meltdown and Spectre vulnerabilities in desktop CPUs can be exploited by malicious code to read from anywhere in memory, exposing both kernel's and other applications private data, which may contain sensitive information such as authentication tokens, passwords, etc. This sent OS vendors rushing to release patches, which early report suggest may cause a performance penalty of up to 11% [12].

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

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

Looking at Figures 5 and 6, we see the cost of providing read safety is higher than write safety. Most applications read from an array or object much more frequently than they write to them. As a result, our VM with read and write safety turned on slows down by 60% on average, corresponding to a 174% slowdown over native C. In addition to bubble sort, MoteTrack also suffers greatly from adding read safety, since it spends 19% of it's instructions reading from the complex RSSI signature data structure. RC5 is the fastest benchmark, since it not only does relatively few array reads and writes, but also spends a large amount of time on expensive variable bit shifts, which have identical performance in both C and AOT compiled versions. The result is a slowdown of only 33% compared to native C for the fully safe version.

Keeping heap bounds in registers In Section **??** we considered several alternatives to our chosen heap bounds check, one of which was to keep the bounds in dedicated registers

to avoid having to fetch them from memory for each check. Here we evaluate this choice.

The expected performance for this alternative is easily calculated. Having the bounds in registers would reduce the cost of the check from 22 to 14 cycles, reducing the safety overhead by $8/22 = 36\%$. However, this uses 4 registers which we can't use for stack caching.

To estimate the performance of this approach, we ran our experiments a second time for our unsafe VM, this time reducing the number of registers available to the stack cache by 4. Since this doesn't affect the number of heap accesses, we then added the observed overhead for safety checks, reduced by 36%.

Figure 7 shows the overhead for our chosen approach with the heap bounds in memory, compared to the expected overhead when the heap bounds are stored in registers. For some benchmarks such as bubble sort and MoteTrack, the savings in heap bounds checks outweighs the reduced effectiveness of the stack cache. But the improvement in performance is relatively small, and for other benchmarks the reverse is true, showing minor slowdowns when heap bounds are kept in registers. On average the benchmarks were quite balanced on average, as was the larger CoreMark benchmark.

As future work we may consider using some basic statistics, such as the percentage of array write instructions and average stack depth, to choose one of the two options on a per-method basis. But as usual there is a tradeoff, in this case VM size and complexity, which may not be worth the effort given the relatively small gains.

0.4.2 Code-size cost

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

Most of our checks are not more complex than comparing two integers, and failing if a condition is not met. The most complex part is deciding the stack effects of instructions to guard against stack over- or underflow. This comes in the form of a table that encodes the effects of most instructions, and some specialised code to analyse a handful of instructions without fixed effect. In total, the increase in VM size for our safe version is a modest 1776

bytes.

As we can see in Table 4, the size of the code the VM generates increases by only $(181.8/177.7) - 1 = 2.3\%$. Since most checks occur at load-time, most instructions produce exactly the same native code in the safe version of our VM. The exceptions are `INVOKEVIRTUAL` and `INVOKEINTERFACE`, which now contains the expected stack effects to realise check **??**, and the array and object write instructions `PUTFIELD` and `PUTARRAY`, which now emit a single extra `CALL` instruction the `heapcheck` routine. Since these instructions are both relatively rare, and already generate a larger block of native instructions than most instructions do in the unsafe version, the total effect on code size is very limited.

0.4.3 Comparison to native code alternatives

As discussed in Section **??**, several non-VM approaches have been proposed to guarantee safety on a sensor node. Two of these, *t-kernel* and Harbor, allow the node to guarantee safety independent of the host. In this section we compare these to our approach, and consider the question whether a VM is a good way to provide safety.

t-kernel reports a slowdown of between 50 and 200%, which is roughly in the same range as our VM. However both *t-kernel* and our approach provide additional advantages. In *t-kernel*'s case a form of virtual memory, and for our VM platform independence. This makes them hard to compare, but we note that while the performance of both systems is similar, *t-kernel*'s code size overhead is much larger at a 6-8.5x increase, limiting the size of programmes we can load onto the device.

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

We implemented this as a benchmark that fills an array of 256 elements with an arbitrary number. This is one of the worst cases for our safe VM since consecutive array

Table 5: Overhead for the fill array benchmark as a percentage of native C

Benchmark	Unsafe overhead	Safe overhead	Increase due to safety checks
8-bit bytes	210.1%	566.4%	356.3%
16-bit shorts	182.8%	451.5%	268.7%
32-bit ints	155.3%	335.6%	180.3%

writes are expensive for two reasons: (i) in our VM this results in repeated executions of the `PUTARRAY` instruction, which calculates the target address for each write, while native code can slide a pointer over the array, eliminating the need for repeated address calculations, and (ii) each of these writes will trigger a call to `heapcheck`.

Our benchmark is implemented as a simple loop as shown in Listing 1.

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

Listing 1: Fill array benchmark (8-bit version)

The resulting overhead is shown in Table 5. We implemented our benchmark for arrays of bytes, shorts and ints. Unfortunately the Harbor paper does not mention the size of elements in their buffer, nor how this would affect their overhead. For our VM, the worst case is filling a byte array because here the relative overhead from address calculation and safety checks is highest.

The worst case extra overhead due to safety checks is 356% of the native C execution time, considerably less than Harbor, and dropping to 180% when storing ints instead of bytes. In addition, our VM also incurs overhead not related to safety checks, but the total overhead of 566% in the worst case is still less than half of Harbor's 1230%.

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

The difference is due to Harbor's more fine grained protection, which allows it to grant access to any aligned block of 8 bytes to the application, while our VM's protection is more coarse. If Harbor could be modified to use a check similar to ours, it's overhead

could potentially be reduced to $1230/65 * 22 \approx 416\%$. This puts it in the middle of our three versions in terms of total overhead. However, it is not clear from the paper whether Harbor's architecture could support such a coarse-grained check since it requires all application data that needs run-time write checks to be in a single segment.

While this shows our approach achieves a performance comparable to even an optimised version of Harbor, it does not highlight the advantage of using a VM to provide safety since both approaches much check each write to an array. However, our approach can verify writes to local and static variables to be safe at translation time, eliminating the need for a run-time check, while we can tell from the Harbor sources [17] that its verifier requires *all* stores to go through its write access check. The authors do note that static analysis of the code could reduce the number of checks, but that this would come at the cost of a significantly more complex verifier.

This means our approach should have a distinct advantage for code with more frequent writes to local variables. The Harbor paper also contains an FFT benchmark, and we use the same code from the Harbor sources for our FFT benchmark. In Harbor's case this results in an overhead of 380%. Even with the faster memory access check, this would still result in an overhead of $380/65 * 22 \approx 129\%$, while our VM is significantly faster at only 30.4%.

0.5 Expected perfomance on other platforms

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

While it is impossible to determine exactly what the resulting performance would be on different platforms without porting our VM, we present some results here that indicate

Table 6: Number or registers and word size for the ATMEGA, MSP430, and Cortex M0

	ATMEGA [3, 2]	MSP430 [16, 15]	Cortex M0 [1]
Number of general purpose registers	32	12	13
Word size	8-bit	16-bit	32-bit
Total register file size (bytes)	32	24	52

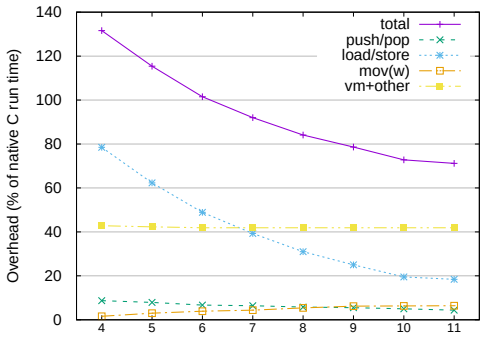


Figure 8: Performance for different stack cache sizes (in pairs of registers)

it is likely to be worse than the results we see on the ATMEGA.

Two parameter that are of important influence to our VM are the number of available registers and the size of the registers. Table 6 lists these parameter for the ATMEGA, and two other common families of embedded CPUS, the Texas Instruments MSP430 and ARM Cortex-M0. These CPUs are similar in many ways, including the amount of RAM and Flash memory typically available, but differ in the number of registers and word size.

0.5.1 Number of registers

We will first look at the impact of the number of available registers. The ATMEGA has 32 8-bit registers available, which we manage as 16 pairs since our VM stores data in 16-bit slots.

Looking at the MSP430, it only has 12 registers, but they are 16-bit. We reserved 5 register pairs on the ATMEGA, leaving us with 11 pairs available for stack caching. If we can achieve the same by reserving 5 16-bit registers on the MSP430, 7 registers will be available to the stack cache.

To evaluate the effect of a smaller number of registers on the stack cache, we reran all our benchmarks while restricting the number of registers the cache manage may use.

Table 7: Performance for different stack cache sizes (in pairs of registers)

Number of register pairs	Overhead push/pop	load/store	mov(w)	vm+other	total
4	8.7	78.5	1.6	42.8	131.6
5	7.9	62.3	3.0	42.3	115.4
6	6.7	48.9	3.9	41.9	101.5
7	6.4	39.3	4.4	41.9	92.0
8	5.8	31.0	5.4	41.9	84.1
9	5.5	25.0	6.2	41.9	78.6
10	5.0	19.5	6.3	41.9	72.8
11	4.4	18.4	6.4	41.9	71.2

Since we need a minimum of 4 pairs for our approach to work, we vary the number of register from 4 to 11.

The results are shown in Figure 8 and Table 7. As is common with caching techniques, the first few registers have the most impact, with half of the performance gain already realised when adding the first two additional registers. Ignoring all other difference for the moment, the effect of reducing the number of register pairs from 11 to 7 is an increase in overhead by 20.8%, to 92%.

The Cortex M0 has one general purpose register more than the MSP430, and at 8 register pairs the overhead drops to 84.1%. In addition, the M0's registers are 32-bit, which means in cases where mostly 32-bit values are used, the stack size is effectively doubled since values can be stored in a single register instead of two pairs o 8-bit registers.

A final interesting thing to note in Figure 8 is the fact the increasing the cache size mostly reduces the load/store overhead. Using only 4 pairs, stack caching has already removed most of the push/pop overhead, which only drops slightly when more registers are used. However, these extra registers reduce load/store overhead significantly since more registers are available for the markloop optimisation, and using more registers it increases the chance that an old, popped value may still be present, allowing popped value caching to eliminate more loads.

0.5.2 Word size

A second important difference between the CPUs in Table 6 is the size of the registers. Our main measure has been the overhead compared to native C performance or code size.

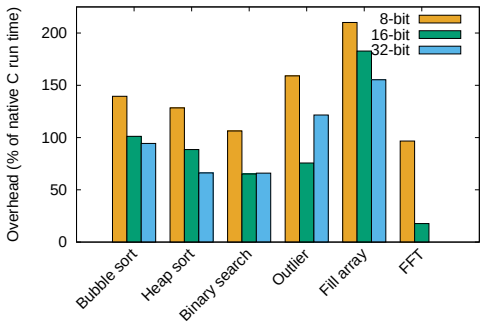


Figure 9: Performance for different data sizes

Table 8: Performance for different data sizes

	8-bit	16-bit	32-bit
Bubble sort	139.5	101.2	94.4
Heap sort	128.4	88.5	66.2
Binary search	106.3	65.2	66.0
Outlier	159.0	75.7	121.6
Fill array	210.1	182.8	155.3
FFT	96.7	17.7	

While having a large register size is good for absolute performance, we expect it to hurt the *relative* performance of our VM.

A number of our benchmarks can be implemented using different data sizes. As mentioned in Section **??**, we used 16-bit data for all of these benchmarks. In Figure 9 and Table 8 we show the resulting performance for 8, 16, and 32-bit versions of these benchmarks (no 32-bit version of the `fix_fft.c` code was available).

In almost all cases, a smaller data size results in a slower performance. The reason is that larger data sizes increase the time spent in the code that both C and AOT versions execute. Taking bubble sort as an example, both versions do the same number of array loads, comparisons, and array stores. When operating on 32-bit data, this takes twice as long as for 16-bit data. In addition the AOT version introduces overhead, in bubble sort's case primarily from calculating array element locations (see Section **??**). 32-bit array access takes 17 cycles, 8 of which are spent on the actual load or store, but for 16-bit array access this is only 4 out of 13 cycles.

This effect is even clearer for more complex operations like multiplication. 16x16 to 16-bit multiplication can be implemented in a few instructions and only takes 10 cy-

cles on the ATMEGA. 16x16 to 32-bit multiplication is implement by calling `avr-gcc`'s `__mulhisi3` function and takes about 50 cycles, while full 32x32 to 32-bit multiplication using `__mulsi3` takes 85 to 100 cycles.

Thus, working with 16-bit or 32-bit data on an 8-bit CPU helps our AOT compiler's relative performance by introducing a larger common component that both C and AOT compiled versions have to execute. On the 16-bit MSP430 or 32-bit M0 this effect will be reduced or eliminated. The exact impact of this is hard to estimate, since using 8-bit data in our VM incurs some an extra overhead because the VM internally stores data as 16-bit values, but it is likely to be in the order of tens of percents of extra overhead.

In Table 8 we see the outlier detection benchmark performs worse for 32-bit data compared to 16-bit data. This is due to a mismatch between the infuser and ProGuard. In JVM byte code, local variables are stored in 32-bit slots. The code generated by `javac` uses a separate slot for each variable, but ProGuard attempts to reduce memory consumption by mapping multiple variables to the same slot if their live ranges do not overlap. The infuser processes the ProGuard optimised code, as shown in Figure **??**, and replaces the JVM's 32-bit operations by 16-bit versions where possible. In the 32-bit outlier detection benchmark, ProGuard mapped a 32-bit and 16-bit variable to the same slot, which prevents the infuser from using 16-bit operations for this variable. This once again highlights the need for a unified, optimising compiler, combining the tasks of `javac`, ProGuard and the Darjeeling infuser.

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

0.6 Limitations and the cost of using a VM

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

Since MyVM is based on Darjeeling, we share many of it's limitations. Like Darjeeling, MyVM does not support multidimensional arrays, reflection, constant data, 64-bit or floating point data types [4]. In addition, MyVM drops support for exceptions and threads since they are much harder to implement in an AOT compiler than in an interpreter. As we will argue in Chapter 1, we feel that if the goal is to provide useful, platform independent, safe, reprogramming of sensor node with adequate performance, instead of simply porting Java to a sensor node, many of Java's more advanced features, especially those that are expensive to implement should be replaced by more lightweight alternatives.

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

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

When optimising code for performance, small choices can often have unexpected consequences. However, we found this to be much more significant when writing Java code for our VM, than when writing the same algorithms in C where `avr-gcc`'s optimisations

often mean two different approaches in C result in similar binary code. An optimising combined compiler and infuser will help a Java developer by doing some of the same optimisations, but there are many examples where the most natural way to solve a problem in Java may not result in the best performance. For example, Suganuma [14] notes that Java code typically results in many small methods and invocations, which can result in a serious performance penalty if they cannot be made lightweight or eliminated by automatic inlining, and allocating many temporary objects hurts performance while creating them once and reusing them usually results in slightly unnatural code. While this is also true for C, the impact of C function calls and allocating temporary objects on the stack is much smaller.

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

Chapter 1

Lessons from JVM

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

One aspect of Java/JVM that makes it an attractive choice for sensor nodes is its simplicity, allowing (a subset of) it to be implement in as little as 8KB [9]. However, it also lacks some features that make it a less good fit for a typical sensor node code in a number of situations. These range from minor annoyances that reduce code readability, to the lack of support for constant data, or high memory consumption for nested data structures that could be a show stopper for a number of applications.

In this chapter we discuss the most pressing issues we encountered, summarised in Table 1.1, and suggest ways they could be improved in future VMs. Although more study is required to turn these suggestions into working solutions, we believe many of the points raised here could be improved with minor changes to Java, leading us to a 'sensor node Java', much like nesC [6] is a sensor node version of C, but some require more drastic changes.

Table 1.1: Point requiring attention in future sensor node VMs

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

Table 1.2: Size of Darjeeling VM components

Component	std.lib (bytes)	VM (bytes)	total (bytes)
Core vm	3529	7006	10535
Strings	8467	1942	10409
Interpreter loop	0	10370	10370
Garbage collection	80	3442	3522
Threads	909	2472	3381
Exceptions	1338	818	2156
Math	222	1274	1496
IO	530	680	1210
Total	15075	28004	43079

1.1 A tailored standard library

A minimum Java APIs for resource constrained devices, the Connected Limited Device Configuration (CLDC) specification, was proposed by Sun Microsystems [11]. The CLDC was primarily intended for devices larger than typical sensor nodes, and not tailored to the characteristics of typical sensor node code. Providing support for the full CLDC specification would require a substantial amount of memory and program space for features that are rarely required for sensor node applications. Table 1.2 shows the code size of library support as implemented in the original Darjeeling VM.

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

On the other hand, the standard library should include abstractions for typical sensor node operations that are missing from the CLDC. The CLDC `Stream` abstraction is intended to facilitate file, network and memory operations. The abstraction is not well suited for communication protocols required by WSN applications, such as I^2C and SPI. In CLDC, connections between devices can be initiated by specifying URI-like strings. However, processing these is relatively expensive, and WSN nodes often identify other nodes using a 16 or 32-bit identifier.

We argue that a tailored library should be designed from the ground-up specifically for sensor node applications. Such a library would include functionality for: (i) basic math; (ii) array operations; (iii) a communication API that encapsulates the low-level protocols typically used (e.g. I^2C); and (iv) a higher-level generic radio and sensor API abstraction.

1.2 Support for constant data

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

This is especially important for arrays of constant data, which are common in WSN applications. When we implement this as a `final` Java array, the compiler emits a static class initialiser that creates a Java array object, and then uses the normal array access instructions to initialise each element individually, as shown in Listing 2.

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

Listing 2: Darjeeling bytecode to initialise a byte array of constant data

There are two problems with this: (i) the array will occupy scarce RAM; and (ii) initial-

ising array elements using JVM instructions requires 4 instructions per element, resulting in 1669 bytes of code to initialise a 256 byte array.

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

FFT The FFT benchmark contains an array of sine wave values that would be too expensive to calculate at run-time. For the 8 bit version, this contains 256 bytes, which is small enough to fit in RAM. However, the 16-bit version contained an array of 1024 16-bit values, which in itself would just fit in memory, but the JVM code to initialise it becomes too large for flash memory, forcing us to use native code to initialise it prior to starting the benchmark.

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

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

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

1.3 Support for nested data structures

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

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

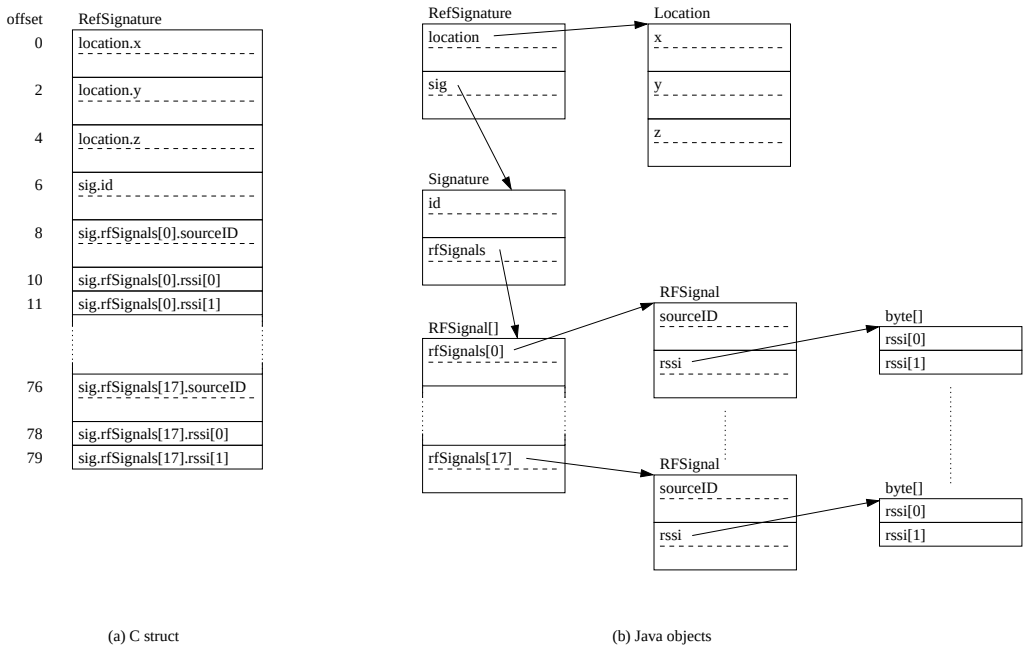
```
#define NBR_RFSIGNALS_IN_SIGNATURE 18
#define NBR_FREQCHANNELS            2

struct RefSignature
{
    Point location;
    Signature sig;
};

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

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

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

Listing 3: MoteTrack `RefSignature` data structure

Since all the arrays are of fixed length, in C the layout of the whole structure is known at compile time, shown in Figure 1.1.

In Java, we have no concept of a struct. As described in Section **??**, in Java every object is made up of a list of primitive values: either an int or a reference to another object. Thus, the most natural way to translate the C structures in Listing 3 to Java, is as a collection of objects and arrays on the heap, as shown in the right half of Figure 1.1. Note that every one of the 18 `RFSignal` structs becomes an object, which in turn has a pointer to an array of rssi values.

There are two problems with this. First, since the location of these Java objects is



(a) C struct

(b) Java objects

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

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

The second problem is the added memory usage. The C struct only takes up 80 bytes, all used to store data. The Java version allocates a total of 40 objects, 36 of which are spent on the `RFSignal` objects and their arrays of RSSI values. Each of these requires a heap header, which takes up 5 bytes in Darjeeling. In addition, the 19 arrays have a 3 byte header, and the Java objects and arrays contain a total of 39 references, which take up 2 bytes each. In total, the collection of Java objects takes up $80+40*5+19*3+39*2 = 415$ bytes.

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

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

1.4 Better language support for shorts and bytes

Because RAM is scarce, 16-bit short and single byte data types are commonly used in sensor node code. The standard JVM only has 32 and 64-bit operations, and variables and stack values are stored as 32-bit, even if the actual type is shorter. On a sensor node this wastes memory, and causes a performance overhead since most nodes have 8-bit or 16-bit architectures, so many sensor node JVMs, including Darjeeling, introduce 16-bit operations and store values in 16-bit slots.

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

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

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

Passing literal integer values to a method call treats them as ints, even if they are short enough to fit in a smaller type, which results in calls like:

```
f((byte)1);
```

While seemingly a small annoyance, in more complex code that frequently uses of shorts and bytes, these casts can make the code much harder to read.

Possible solutions We suggest that C-style automatic narrowing conversions would make most sensor node code more readable, but to leave the option of Java's default behaviour

open, we may implement this as new datatypes: Declaring variable `a` as `unchecked short` would implictly narrow to short when needed, so `a=b+c;` would not need an explicit cast, while declaring it as a normal `short` would.

1.5 Simple type definitions

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

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

Possible solutions At a minimum, we should have a way to define simple aliases for primitive types, similar to C's `typedef`. A more advanced option that fits more naturally with Java, would be to have a strict `typedef` which also does type checking, so that a value of one user defined integer type cannot be accidentally assigned to a variable of another type, without an explicit cast.

1.6 Explicit and efficient inlining

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

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

This macro is called in four places, and is very performance critical. Tools like Proguard [13] can be used to inline small methods, but in this case it is larger than Proguard's size threshold. This leaves developers with two unattractive options: either leaving it as a method and accepting the performance penalty, or manually copy-pasting the code, which is error-prone and leads to code that is harder to maintain.

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

Another option is to give the developer more control over inlining, which could easily be achieved by adding an `inline` keyword to force the compiler to inline important methods. These annotations are usually placed at the method level.

1.7 An optimising compiler

As discussed in previous chapters, but listed here again for completeness, Java compilers typically do not optimise the bytecode but translate the source almost as-is. Without a clear performance model it is not always clear which option is faster, and the bytecode is expected to be run by a JIT compiler, which can make better optimisation decisions knowing the target platform and runtime behaviour. However, a sensor node does not have the resources for this and must execute the code as it is received. This leads to significant overhead, for example by repeatedly reevaluating a constant expression in a loop.

Possible solutions Even without a clear performance model, some basic optimisations can be done. The results in Section **??**, show that some very conservative optimisations can already result in code twice as fast as the original. These could be further expanded, and combining the tasks of the optimiser and infuser can further improve performance as shown in Section 0.5.2.

1.8 Allocating objects on stack

In Java anything larger than a primitive value has to be allocated on the heap. This introduces a performance overhead, both for allocating the objects, and the occasional run of the garbage collector, which may take several thousand cycles.

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

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

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

Listing 4: Avoiding multiple object allocations in the LEC benchmark

This overhead can often be reduced by allocating earlier and reusing the same objects in a loop. In Listing 4 we see this implemented for the LEC benchmark, where we use the `bsi` object to return two values from `encode` to `compress`. Instead of creating a

new object in each iteration in the `compress` method where it is needed, we create `bsi` once, outside of the main loop, and pass the same object to `compress` multiple times.

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

Possible solutions On desktop JVMs, escape analysis [5, 7] is used to determine if an object can be safely allocated on the stack instead of the heap, thus saving both the cost of heap allocation, and the occasional garbage collection run triggered by it.

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

There is a risk to doing this automatically. In our sensor node VM, the split between heap and stack memory is fixed, and both are limited. If the compiler automatically puts all objects that could be on the stack in the stack frame instead of the heap, we may end up with an empty heap, and stack overflow. Therefore, it is better to leave this optimisation to the develop by also introducing a new keyword at the language level, so the developer can explicitly indicate which objects go on the stack and which in the heap. Of course escape analysis is still necessary to check at compile time this keyword is only used in places where the object can go on the stack.

1.9 Reconsidering advanced language features

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

While features like threads and garbage collection are all useful, they come at a cost. The trade-off for a sensor node VM is significantly different from a desktop VM: many of Java's more advanced features are vital to large-scale software development, but the size of sensor nodes programmes is much smaller. And while VM size is not an issue on the desktop, these features are relatively expensive to implement on a sensor node. We believe a VM developed from scratch, with the aim of providing platform independence, safety, and performance through AOT compilation, would end up with a design very different from JVM.

In Table 1.2 we show the code size in Darjeeling for some features we discuss below. These were determined by counting the size of functions related to specific features. The actual cost is higher since some, especially garbage collection, also add complexity to other functions throughout the VM. Combined, the features below and the string functions mentioned in Section 1.1 make up about half the VM.

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

1.9.1 Threads

As shown in Table 1.2, support for threads costs about 10% of the VM size, if we exclude the string library. In addition, there is the question of allocating a stack for each thread. If we allocate a fixed block, it must be large enough to avoid stack overflows, but too large a block wastes precious RAM. Darjeeling allocates each stack as a linked list of frames on

the heap. This is memory efficient, but allocating on the heap is slower and occasionally triggers the GC.

A more cooperative concurrency model is more appropriate for sensor nodes, where lightweight tasks voluntarily yield the CPU and share a single stack. This is also the approach to concurrency chosen by a number of native code systems, including *t-kernel* [8], nesC [6], and more recently Amulet [10].

1.9.2 Exceptions

In terms of code size, exceptions are not very expensive to implement in an interpreter, but they are hard to implement in an AOT compiler. We also feel the advantage of having exceptions is much lower than the other features mentioned in this section. They could be easily replaced with return values to signal errors.

1.9.3 Virtual methods

It is hard to quantify the overhead of implementing virtual methods since the code for handling them is integrated into several functions. In terms of size it is most likely less than 2KB, but the performance overhead is considerable. The target of a virtual method call must be resolved at runtime, they cannot be made lightweight, and an AOT compiler can generate much more efficient code for calls to static methods.

In practice we seldom used virtual methods in sensor node code, but some form of indirect calls is necessary for things like signal handling. It should be possible to develop a more lightweight form of function pointers that can be implemented efficiently. However, the details will require more careful study.

1.9.4 Garbage collection

Finally, garbage collection is clearly the most intrusive one to change. While the first three features could be changed with minor modifications to Java, the managed heap is at its very core.

Still, there are good reasons for considering alternatives. Table 1.2 shows the GC methods in Darjeeling add up to about 3.5KB, but the actual cost is much higher as many other parts of Darjeeling are influenced by the garbage collector.

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

1.10 Towards better sensor node VMs

In this chapter we described a number of issues we encountered over the years while using and developing sensor node VMs. They may not apply to every scenario, but the wide range of the issues we present suggests many applications will be affected by at least some.

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

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

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

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

Chapter 2

Conclusion

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

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

1. After identifying the sources of overhead in previous work on ahead-of-time compilers for sensor nodes, MyVM introduced a number of optimisations to reduce the performance overhead by over 80%, and the code size overhead by 60%, resulting in an average performance overhead of 105%, and the code size overhead of 82% compared to native C.

   The optimisations introduced by MyVM 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.

   The price to pay for platform independence and a safe execution environment comes in three forms. We believe the remaining performance overhead will be acceptable for many applications, but the increase in code size, and the space taken by the VM do limit the size of applications we can load onto a device. In addition, the overhead

in memory usage was a problem for a number of benchmarks.

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

   This results in a modest overhead in terms of VM size, and the fact that most checks are performed at translation time means the additional overhead for providing safety is limited to a 35% slowdown and 6% increase in code size, again relative to native C.

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

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

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

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

Table 2.1: Comparison of our approach to related work

Approach	Platform indep.	Safe	Performance	Code size
Native code	No	No	1x	1x
Interpreters	Yes	Mostly no	300-23000% slower	50% smaller
Ellul's AOT	Yes	No	123-844% slower	120-346% larger
Safe TinyOS	No	Yes[a]	17% slower	27% larger
t-kernel	No	Yes	50 to 200% slower	500 to 750% larger
Harbor	No	Yes	160 to 1230% slower	30 to 65% larger
Our VM (unsafe)	Yes	No	71% slower	78% larger
Our VM (safe)	Yes	Yes	105% slower	82% larger

[a] requires a trusted host

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

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

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

However our results show that our approach is on-par with or faster than the two existing native code approaches, and provides platform independence at the same time. We also note that if we require a platform independent way of reprogramming our nodes, making it safe comes at a relatively low extra cost, with our safe VM only 19.9% slower than the unsafe version.

Appendix A

LEC benchmark source code

Appendix B

Outlier detection benchmark source code

Appendix C

Heat detection benchmark source code

C.0.1 Calibration

C.0.2 Detection

Bibliography

[1] ARM. Cortex M0 Technical Reference Manual. 2009.

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

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

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

[5] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape Analysis for Java. *ACM SIGPLAN Conference on Object- Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Nov. 1999.

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

[7] B. Goetz. Java theory and practice: Urban performance legends, revisited, Sept. 2005.

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

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

[10] J. Hester, S. Lord, R. Halter, D. Kotz, J. Sorber, T. Peters, T. Yun, R. Peterson, J. Skinner, B. Golla, K. Storer, S. Hearndon, and K. Freeman. Amulet: An Energy-

Efficient, Multi-Application Wearable Platform. In *The 14th ACM Conference on Embedded Networked Sensor Systems (SenSys 2016)*, 2016.

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

[12] N. A. Simakov, M. D. Innus, M. D. Jones, J. P. White, S. M. Gallo, R. L. DeLeon, and T. R. Furlani. Effect of Meltdown and Spectre Patches on the Performance of HPC Applications. 2018. arXiv:1801.04329.

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

[14] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1), 2000.

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

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

[17] UCLA Networked & Embedded Systems Laboratory. Sos operating system, 2009.