Run-time Compilation of Bytecode in Sensor Networks

Joshua Ellul
School of Electronics and Computer Science
University of Southampton
Southampton, United Kingdom
Email: je07r@ecs.soton.ac.uk

Kirk Martinez
School of Electronics and Computer Science
University of Southampton
Southampton, United Kingdom
Email: km@ecs.soton.ac.uk

***Abstract*—Recent work on virtual machines for wireless sensor networks has demonstrated the benefits of using a Java programming paradigm for resource constrained sensor networks. Results have shown that a virtual machine approach greatly suffers from interpretation overheads. In this paper, we present run-time compilation of bytecode in wireless sensor networks which leverages from both a compact platform independent bytecode program representation as well as a native execution platform for efficient execution of code. Our results show that run-time compilation provides a substantial decrease in execution overheads when compared with an interpreter.**

***Keywords*-Java; Compilers; Sensor Networks; Bytecode;**

I. Introduction

Wireless sensor networks present programming challenges that are not present in traditional computing systems. The networks usually consist of many small battery powered devices that are intended to operate for months or years with minimal computation and storage resources. The learning curve involved to begin development for sensor networks is quite steep, since not only does one have to understand the embedded systems paradigm, but new languages and tools must be adopted.

Recent work on Java compatible virtual machines [1][2] attempt to alleviate the paradigm shift by allowing developers to create Java applications which are then converted to bytecode and interpreted on the sensor nodes. Results presented in [1] demonstrate the costs of an interpreted virtual machine, being high execution costs.

When Java virtual machines for traditional hardware were becoming more popular the costs of interpretation were realised and initiatives to compile bytecode to native code began [3]. However, it is widely assumed that compiling bytecode to native code on such resource constrained devices as sensor nodes is either impossible or infeasible due to the limited storage and memory availability [4][5][6][7].

Our primary contribution is that we demonstrate that run-time compilation of bytecode to native code is both feasible and practical, and a substantial decrease in execution overhead can be achieved with a simplistic compilation model.

The remaining of this paper is split up as follows. Section II provides an overview of related work in the virtual machine resource constrained devices community as well as techniques which have been proposed for higher end hardware platforms. Design requirements, choices and strategies are explained in Section III. In Section IV we demonstrate an implemention of a run-time compiler on a MSP430F1611 microcontroller on a Telos B sensor node. We evaluate how our approach compares with recent work on virtual machines for sensor networks in Section V and then follow with final thoughts and a pathway for future work in Section VI.

II. Related Work

Our work is centered in the sensor network virtual machine and Just-in-Time compilation research areas. We will now provide background information on the different research areas.

A. Virtual Machines

Maté [8] is a well known virtual machine for sensor networks. It is a stack based virtual machine that was later included in TinyOS under the name Bombilla. Maté interprets virtual machine scripts that are encoded in capsules. The VM restricts the programmer to eight code capsules of which each capsule cannot consist of more than 24 instructions. Maté scripts are comprised of assembly-like single byte instructions that expose higher level functionality. Due to the limited amount of code capsules and size of each code capsule, the Maté virtual machine restricts what type of applications can be implemented in the virtual machine scripts [9].

Darjeeling [1] and Mote Runner [2] are two recent initiatives at creating a Java compatible virtual machine for wireless sensor networks. In principle, the two virtual machines are very similiar. They both provide a compilation process which translates source code to bytecode and then to an intermediate bytecode which is then interpreted by the virtual machine. The virtual machines both implement a 16-bit stack operand sized instruction set. Primary differences include a threading model versus a reactive programming model using event callbacks. The authors of [2] claim that implementing threads in such a resource constrained virtual machine has some drawbacks. This roots to the fact that the stack assigned to each thread has to be large enough to support arbitrary user applications otherwise the virtual machine must provide a dynamic stack mechanism. This means that threads can be implemented with either memory wastage due to assigning too large of a stack space or a larger footprint required to implement a dynamic sized stack scheme. Besides this the virtual machine must expose thread synchronization features which again will increase the footprint marginally. Darjeeling, in turn implements dynamic sized thread stacks.

Benchmark tests presented in [1] highlight the costs of interpreting bytecode. A penalty of a factor in the range between 30 to 113 times native code execution is heeded. This is where our contribution is intended to fit in by decreasing the overhead of executing bytecode in sensor networks.

B. Just-in-Time Compilation

Just-in-Time (JIT) compilation techniques although assumed to be a modern invention which was developed for Java have been around since the 1960s [10] with JIT compilation for LISP [11]. Over the years JIT compilation has evolved and resurfaced many times for different architectures and requirements including Smalltalk [12], Prolog [13] and many others. However, the main underlying goal of JIT compilation, or rather run-time compilation, is to translate from a higher level representation code source to a lower level more efficiently undertstood native code target.

More recently JIT compilation techniques have reermerged with the popularity of Java. However, when Java was still premature, the costs of interpreting its bytecode was notable. The quote from [14] sums up the state of affairs that Java was in, ”Java isn’t just slow, it’s really slow, surprisingly slow.”

Thus, a high amount of iniative was put into JIT compilation techniques for Java, and as processing power become more and more abundant, more complex strategies could be implemented to obtain more efficient executable code [15][16][17]. It is probably for this reason that it is so widely accepted that run-time compilation for resource constrained devices is either impossible or impractical given the resource constraints[4][5][6][7].

III. Design

A Java programming paradigm can provide many advantages for programming wireless sensor networks, however recent work on compatible virtual machines in [1] has revealed the high execution costs incurred when using an interrpreter. Therefore, we have designed a run-time compiler which translates bytecode into the required platform’s native code.

A. Compilation Process

We propose using a three stage compilation process, as depicted in Figure 1. The first stage involves compiling

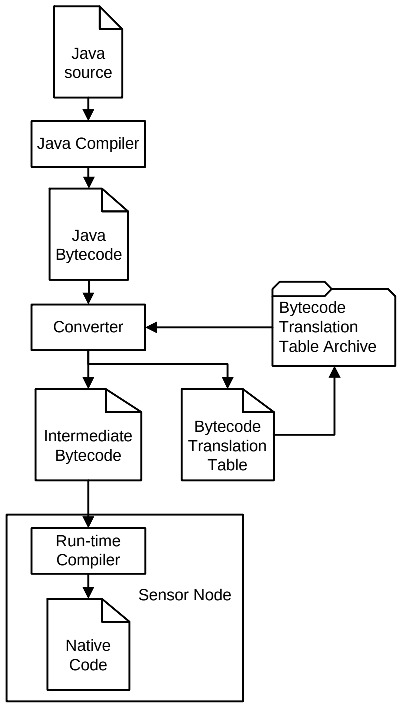

Figure 1. The translation process from Java source code down to native code compiled on the sensor node.

Java source code to Java bytecode using the standard Java compiler or any other Java compiler. Next, the Java bytecode produced will be passed through the converter which will convert the Java bytecode into our modified intermediate bytecode. This process involves resolving constants, class names and function signatures to a smaller symbolic representation. All external class and function references are resolved against an archive of bytecode translation tables which stores the translation information for previously converted classes. The converter also produces a bytecode translation table which is later used to resolve references to the class when referenced from another class. The intermediate bytecode can then be transferred to sensor nodes which will then be compiled to native code by the run-time compiler.

B. Run-time Compiler and Execution

The run-time compiler is responsible for generating native code from the intermediate bytecode, and for loading and unloading classes and functions from program memory. Java bytecode is stack based, and to minimize the footprint of the run-time compiler, we propose to make use of a run-time operand stack which mimics the Java bytecode operand stack. We believe that we can achieve a substantial decrease in execution overhead without introducing any compiler optimizations, since we plan to remove the high interpretation overhead.

1) The Run-time Operand Stack and Stack Frame Layout: As mentioned above, our run-time operand stack will mimic

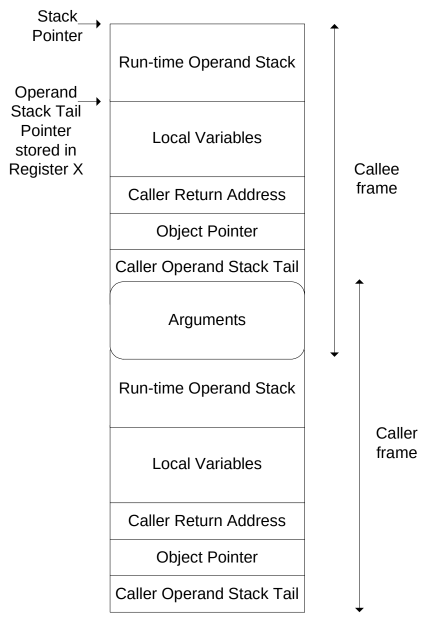

Figure 2. Run-time stack frame layout.

a stack maintained by an interpreter, except for the fact that the stack is natively *pushed* and *popped* to. Such a simplistic mapping from bytecode instructions to native instructions which operate on the microcontroller's stack will allow our run-time compiler to both consist of a small footprint whilst also minimizing execution overhead for a stack based bytecode virtual machine compared with an interpreter such as Darjeeling and Mote Runner.

We have chosen to implement a typical Java virtual machine stack frame scheme as shown in 2. Our proposed run-time compilation technique can be altered to use other stack frame layouts if deemed necessary, however, the contiguous stack frame layout is appropriate for the design choices we have made, in that our proposed execution environment uses a reactive programming model rather than a threaded one.

The stack frame consists of the caller operand stack tail, i.e. a pointer to where the run-time operand stack ends and the local variables begin for the caller. By maintaining such a pointer to the end of the run-time operand stack functions can effortlesly access local variables and arguments, since the operand stack dynamically grows and shrinks. A pointer to the callee function's tail of the run-time operand stack is displayed as well with an indication that it is stored in Register X, which may be any microcontroller register which is suitable. The stack pointer in Figure 2 refers to the microcontroller's current position of the stack pointer.

The object pointer is a pointer to the current object instance, and the caller return address is the memory address where execution should continue once the executing function has returned. A local variable slot is statically sized to the required amount on entrance into functions.

The run-time operand stack as previously mentioned is the bytecode operand stack which is natively *pushed* and *popped* to. Function arguments are essentially part of the run-time operand stack which the caller and callee share by *pushing* to and *popping* from respectively.

2) Compiling Bytecode Instructions to Native Code: The run-time compiler traverses the bytecode received and translates it into native code. This is performed by inspecting the bytecode in two passes. The first pass is used to determine the number of local variables. This knowledge then allows the run-time compiler to generate native code which can directly access the local variables and the other stack frame elements by keeping track of the run-time operand stack tail in one of the microcontroller's registers as discussed in Section III-B1.

No native code is generated in the first pass. The second pass then converts each bytecode instruction into corresponding native code commands which mimic operations on the microcontroller stack to emulate the bytecode operand stack. Jump and goto instructions are converted to native code instructions in the second phase but the jump addresses are not generated in the second pass since not all jump locations will be able to be resolved. Generated native code is built in RAM rather than program memory, since flash block writing is more efficient than single byte or word writes. Once, the second pass of the bytecode is complete all possible jump locations are resolveable and thus, the missing jump memory addresses are filled in. Finally, the native code is written to program memory. We describe further the implementation of the MSP430 run-time compiler in Section IV.

IV. Implementation

We have implemented a run-time compiler for the MSP430F1611 and have tested it on the Telos B [18] sensor node. As discussed in the Design section of this paper, the run-time operand stack and the native code stack operations that mimic the operations an interpreter would perform is the central building block that enables us to achieve a small footprint run-time compiler.

Just by removing the interpretation element of the virtual machine we believe that substantial execution gains can be achieved. Implementations from one platform to another of our run-time compiler will only differ in the actual bytecode to native conversions, but all the other logic is essentially the same. Thus, in this section we need only demonstrate the bytecode to MSP430F1611 native code conversions implemented in our run-time compiler as all other implementation details are trivial and in accordance with the Design section of this paper.

The second bytecode inspection pass as described in Section III-B2 converts bytecode to native code. Table I displays different bytecode instructions, their respective generated native code and the effect on the run-time operand

Table I
BYTECODE TO NATIVE CODE CONVERSIONS

Bytecode	Native Code	Stack [before] → [after]
iload_0	PUSH <OFFSET>(R11)	→ value
iconst_0	PUSH #0	→ 0
aload_0	PUSH <OFFSET>(R11)	→ objectref
getfield	POP.W R9 MOV.W @R9,R9 PUSH <OFFSET>(R9)	objectref → value
putfield	POP R10 POP R9 MOV.W @R9,R9 MOV.W R10,<OFFSET>(R9)	objectref, value →
istore_0	POP R10 MOV.W R10,<OFFSET>(R11)	value →
iinc	ADD.W <CONST>,<OFFSET>(R11)	→
iadd	POP R10 POP R9 ADD.W R10,R9 PUSH R9	value1, value2 → result
if_icmpeq	POP R9 POP R10 CMP.W R9,R10 JEQ (<JUMP ADDR>)	value1, value2 →
dup2	PUSH 0x0004(SP) PUSH 0x0004(SP)	value2, value1 → value2, value1, value2, value1
putstatic	POP R10 MOV.W R10, <STATIC VAR ADDR>	value →

stack. We have only provided the conversion for some of the bytecode instructions due to brevity. As can be seen from the generated native code, a high amount of native code `PUSH` and `POP` commands will exist throughout the generated native code application. In fact, the resultant push and pop operations should tally the amount of operations an interpreter would execute. The primary difference is that the push and pop operations generated by our run-time compiler do not incur any interpretation overheads. More so, we have removed all interpretation overhead from the execution paradigm.

We have decided to use the register R11 as the operand stack tail pointer, which will facilitate direct access to the local variables on the stack frame as described in Section III-B1. We then make use of the other registers in descending order from R10. The reason why we have chosen to use these registers is that, the compiler's function calling convention is to pass parameters in the order from R12 to R15 and then R4 to 11. Thus, since we would like to leave options open for the run-time compiled code to be able to interact with precompiled C native code, we can ensure that at least the first 9 parameters can be passed without having to check whether the register was being used for a stack operation. However, this is beyond the scope of this paper and is left for future work.

Let's consider compiling the bytecode for a Java assignment `a = b + c`, where the Java compiler has reference a, b and c by 0, 1 and 2 respectively. This will be compiled to the following bytecode:

```
    iload_1
    iload_2
    iadd
    istore_0
```

The run-time compiler will then produce the following native code commands for the above generated bytecode:

```
    ; iload_1
0   PUSH <OFFSET TO VARIABLE 1>(R11)

    ; iload_2
1   PUSH <OFFSET TO VARIABLE 2>(R11)

    ; iadd
2   POP R10
3   POP R9
4   ADD.W R10,R9
5   PUSH R9

    ; istore_0
6   POP R10
7   MOV.W R10,<OFFSET TO VARIABLE 0>(R11)
```

The `iload_1` bytecode instruction is translated to a native *pushing* of the variable referenced by 1, i.e. b. The same applies for the loading of variable 2. The `iadd` bytecode instruction results in two native code *poppings* into registers 10 and 9 which are then added together and the result stored in register 9 by the `ADD.W` native command. The result, to mimic the bytecode operand stack, has to be put on the run-time operand stack, and thus, the result which was stored in register 9 is pused onto the run-time operand stack. Finally, the result is set to be stored into variable 0, i.e. a, by *popping* the result just pushed onto the stack into register 10 and then moving the value of register 10 to the memory position of variable 0.

As can be seen from the example above, the PUSH performed at line 5 and the following POP could be eliminated, and the value of register 9 could be directly moved to variable 0's memory position. However, we have decided to allow such extra PUSH and POP operations so as to minimize the run-time compiler's footprint. The optimization of generated native code is an area of research which we leave to future work, since this involves carefully analyzing the gains achieved against the increase in footprint. We will now move on to an evaluation of our approach to see exactly what gains are achieved by using a run-time compiler, and to see whether mimicing the bytecode operand stack is a good idea.

V. Evaluation

In order to evaluate our approach we have implemented some benchmark tests as used in [1]. We have implemented the benchmarks using the same Java code from the Darjeeling distribution, just with minor changes to remove some Darjeeling specific function calls. We have implemented both a 16 and 32 bit bubble sort test and an MD5 test. We did not implement the vector convolution test as we could not locate it in the Darjeeling distribution.

The bubble sort test sorts 256 integer values which are initialised in descending order for 16 and 32 bit integer types and operations respectively. The MD5 performs 1,000 MD5 hashes on each of the strings 'a', 'abc', 'darjeeling' and 'message digest'.

The results for our benchmark tests are shown in Table II. The C column displays the amount of time taken to run the benchmarks using native C. The RTC column displays the time taken to execute the benchmarks using our run-time compiler, and the DVM column shows the time taken to execute the tests on the Darjeeling platform. The RTC/C and DVM/C columns display the ratios between the time taken on the natice C implementation against our run-time compiler and the Darjeeling virtual machine respectively.

Undoubtedly, a direct comparison between our benchmark implementation and the Darjeeling benchmark implementation does not provide a full correlation, since we have implemented our tests on a 16 bit MSP430 microcontroller, whilst the Darjeeling tests were implemented on an 8 bit ATmega128 microcontroller, both microcontrollers running at 8MHz. However, the large differences in the benchmark results show that even if the Darjeeling virtual machine were to gain double the speed on a 16 bit microcontroller, still the performance achieved using the Darjeeling virtual machine would be much slower than the performance achieved using run-time compilation.

As can be seen from the results our implementation is only slower by 4.7 to 7.5 times the native C code implementation, whilst the Darjeeling virtual machine ranges from 30.4 to 96.5 times the native code implementation. In the future we will port our run-time compiler to the ATmega128 microcontroller which will allow us to provide a better comparison between the gains achieved using a run-time compiler aginst an interpreter such as Darjeeling.

Table II
Performance Benchmarks

Test	C	RTC	DVM	RTC/C	DVM/C
Bubble sort 16	0.2s	1.5s	19.3s	7.5	96.5
Bubble sort 32	0.3s	1.8s	23.3s	6	77.7
MD5	13.1s	61.6s	399.7s	4.7	30.4

We do not provide any bytecode size analysis since this is really beyond the scope of our work. Our run-time compiler will operate on bytecode similiar to that used in interpreters such as Darjeeling, so effectively the size of the bytecode which is distributed in a run-time compiler equipped sensor network will be the same as an interpreter equipped one. Anyhow, this is an area of interest which we will look into in the future by generating optimized bytecode offline at the converter stage of our compilation process which occurs outside of the sensor network.

VI. Conclusion

Recent work on virtual machine interpreters has demonstrated the benefits of being able to develop sensor network applications using Java. However, the benefits come with some a drawback as well, being inefficient execution of code. It is widely assumed that compiling bytecode to native code on resource constrained devices such as sensor nodes is either impossible or impractical due to the limited storage and memory space. In this paper, we have shown that it is both possible and feasible to implement a simplistic run-time compiler that provides an execution environment which is substantially more efficient than an alternative interpreted method.

We have provided an evaluation of our implemented run-time compiler on a MSP430F1611 microcontroller on a TelosB sensor node and have shown for some benchmark applications that such an implementition incurs an execution cost that is between 4.7 to 7.5 times the amount of natively compiled implementations. We have compared the performance of the run-time compiler with a recently presented Java compatible virtual machine interpreter, namely Darjeeling, in which we demonstrate that a simplistic run-time compiler can achieve a substantial performance increase whilst still maintaining a minimal footprint. Thus, although a run-time compiler involves slightly more work to be port to different platforms than an interpreter, the results heeded definitely justify this.

A. Future Work

Our work on the run-time compiler opens up many research questions for resource constrained device compilation techniques. We plan on continuing our research with the following items:

- A lot of work on offline Java bytecode optimization exists and it would be useful to view the benefits of using optimized bytecode alongside a run-time compiler.
- Debugging of bytecode on a virtual machine running on resource constrained devices can prove to be very difficult. We would like to look into reverse native code to bytecode techniques in aim of achieving direct debugging of bytecode on sensor node hardware.

Acknowledgment

The authors would like to thank the Government of Malta for supporting this research through the Malta Government Scholarship Scheme ME 367/07/7.

References

[1] N. Brouwers, K. Langendoen, and P. Corke, “Darjeeling, a feature-rich VM for the resource poor,” in *SenSys09*, Berkeley, CA, nov 2009. [Online]. Available: http://www.es.ewi.tudelft.nl/papers/2009-Brouwers-darjeeling.pdf

[2] A. Caracas, T. Kramp, M. Baentsch, M. Oestreicher, T. Eirich, and I. Romanov, “Mote runner: A multi-language virtual machine for small embedded devices,” *Sensor Technologies and Applications, International Conference on*, vol. 0, pp. 117–125, 2009.

[3] 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 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1996, pp. 90–99.

[4] D. Palmer, “A virtual machine generator for heterogeneous smart spaces,” in *VM’04: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*. Berkeley, CA, USA: USENIX Association, 2004, pp. 1–1.

[5] J. Koshy and R. Pandey, “Vmstar: Synthesizing scalable runtime environments for sensor networks,” in *In In Proceedings of the third international Conference on Embedded Networked Sensor Systems (Sensys*. ACM Press, 2005, pp. 243–254.

[6] B. L. Titzer, J. Auerbach, D. F. Bacon, and J. Palsberg, “The exovm system for automatic vm and application reduction,” in *PLDI ’07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2007, pp. 352–362.

[7] R. Pandey and J. Koshy, “A software framework for integrated sensor network applications,” in *InterSense ’06: Proceedings of the first international conference on Integrated internet ad hoc and sensor networks*. New York, NY, USA: ACM, 2006, p. 11.

[8] P. Levis and D. Culler, “Maté: a tiny virtual machine for sensor networks,” *SIGOPS Oper. Syst. Rev.*, vol. 36, no. 5, pp. 85–95, 2002.

[9] N. Reijers and K. Langendoen, “Efficient code distribution in wireless sensor networks,” in *WSNA ’03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*. New York, NY, USA: ACM, 2003, pp. 60–67.

[10] J. Aycock, “A brief history of just-in-time,” *ACM Comput. Surv.*, vol. 35, no. 2, pp. 97–113, 2003.

[11] J. McCarthy, “Recursive functions of symbolic expressions and their computation by machine, part i,” *Commun. ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[12] L. P. Deutsch and A. M. Schiffman, “Efficient implementation of the smalltalk-80 system,” in *POPL ’84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. New York, NY, USA: ACM, 1984, pp. 297–302.

[13] R. C. Haygood, “Native code compilation in sicstus prolog,” in *Proceedings of the eleventh international conference on Logic programming*. Cambridge, MA, USA: MIT Press, 1994, pp. 190–204.

[14] P. Tyma, “Why are we using java again?” *Commun. ACM*, vol. 41, no. 6, pp. 38–42, 1998.

[15] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko, “Compiling java just in time,” *Micro, IEEE*, vol. 17, no. 3, pp. 36–43, May/Jun 1997.

[16] B.-S. Yang, S.-M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioglu, and E. Altman, “Latte: A java vm just-in-time compiler with fast and efficient register allocation,” *Parallel Architectures and Compilation Techniques, International Conference on*, vol. 0, p. 128, 1999.

[17] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani, “Overview of the ibm java just-in-time compiler,” *IBM Syst. J.*, vol. 39, no. 1, pp. 175–193, 2000.

[18] J. Polastre, R. Szewczyk, and D. Culler, “Telos: enabling ultra-low power wireless research,” in *IPSN ’05: Proceedings of the 4th international symposium on Information processing in sensor networks*. Piscataway, NJ, USA: IEEE Press, 2005, p. 48.