UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL AND APPLIED SCIENCES

Electronics and Computer Science

Run-time Compilation Techniques for Wireless Sensor Networks

by

Joshua Ellul

Thesis for the degree of Doctor of Philosophy

November 2012

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL AND APPLIED SCIENCES
Electronics and Computer Science

Doctor of Philosophy

RUN-TIME COMPILATION TECHNIQUES FOR WIRELESS SENSOR NETWORKS

by Joshua Ellul

Wireless sensor networks research in the past decade has seen substantial initiative, support and potential. The true adoption and deployment of such technology is highly dependent on the workforce available to implement such solutions. However, embedded systems programming for severely resource constrained devices, such as those used in typical wireless sensor networks (with tens of kilobytes of program space and around ten kilobytes of memory), is a daunting task which is usually left for experienced embedded developers.

Recent initiative to support higher level programming abstractions for wireless sensor networks by utilizing a Java programming paradigm for resource constrained devices demonstrates the development benefits achieved. However, results have shown that an interpreter approach greatly suffers from execution overheads. Run-time compilation techniques are often used in traditional computing to make up for such execution overheads. However, the general consensus in the field is that run-time compilation techniques are either impractical, impossible, complex, or resource hungry for such resource limited devices.

In this thesis, I propose techniques to enable run-time compilation for such severely resource constrained devices. More so, I show not only that run-time compilation is in fact both practical and possible by using simple techniques which do not require any more resources than that of interpreters, but also that run-time compilation substantially increases execution efficiency when compared to an interpreter.

Contents

List of Figures

List of Tables

Declaration of Authorship

I, Joshua Ellul , declare that the thesis entitled *Run-time Compilation Techniques for Wireless Sensor Networks* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as: (Ellul and Martinez, 2010b) and (Ellul and Martinez, 2010a)

Signed:.....

Date:.....

Acknowledgements

I would like to thank Dr. Kirk Martinez for his unreserved support, guidance and dedication towards my work throughout the Ph.D.
I would like to thank the Government of Malta for funding the Ph.D. under the Malta Government Scholarship Scheme through award ME 367/07/7.
Thanks also goes out to my family for supporting and encouraging me throughout the Ph.D.

To my family and friends for supporting me throughout the Ph.D. and for being patient and pretending to take an interest in my work :)

Nomenclature

B	battery capacity
$Cycle_s$	length of time for a duty cycle at microcontroller power state
$E(\cdot)$	execution cost function
E_{cycle}	energy consumed for a duty cycle
$E_{exeop}(\cdot)$	execute cost function
$E_{int}(\cdot)$	interpretation cost function
$E_{process}$	energy required to process code
$E_{processbyte}$	energy required to process a byte
E_{repro}	energy required to reprogram code
E_{rx}	energy required to receive code
E_{rxbyte}	energy required to receive a byte
E_s	power consumption at microcontroller power state
$E_{stack}(\cdot)$	stack operation cost function
E_{store}	energy required to store update
$E_{storebyte}$	energy required to store a byte
i	bytecode instruction
I_s	current consumption at microcontroller power state
L	expected lifetime
s	microcontroller power state
S_{rx}	number of bytes of code received
S_{store}	size of update to store
T_s	ratio of time in microcontroller power state

Chapter 1

Introduction

At the dawn of the 21st century, Weiser's (1999) vision that computing will become ubiquitous and invisible had seen its first steps towards realisation with the proposal of Wireless Sensor Networks (WSNs) and Smart Dust (Warneke et al., 2001). Many promising WSN applications were proposed for military (Arora et al., 2004), habitat monitoring (Mainwaring et al., 2002) and environmental (Martinez et al., 2004) use. Pister (2001) envisaged that by 2010, wireless sensor nodes would cost a dollar in large volumes. Although wireless sensor nodes have not yet been made available at this price (for whatever reason), many will agree that the successful adoption of wireless sensor networks is highly dependent on cost. That said, with sensor node hardware costs estimated to eventually be in the range of a few dollars, actual firmware development may comprise much of the cost involved in deploying a sensor network.

Wireless sensor networks present programming challenges that are not present in traditional computing systems. The typical memory resources available on sensor nodes are extremely limited, typically equipped with tens of kilobytes of program space and ten kilobytes of volatile memory. Therefore, algorithm developers must ensure that memory is not used imprudently. More challengingly, typical applications impose strict prolonged lifetimes whilst sensor nodes are deployed with extremely limited power resources. Therefore, software developers must ensure that algorithms are highly efficient and ensure that the processor and any peripheral hardware are put into sleep modes as much as possible. Sensor nodes require to communicate with other nodes in order to relay sensed information or else to send updates or configuration messages into the network. Therefore, sensor nodes usually cannot sleep indefinitely however are required to follow wakeup schedules so that nodes can communicate in predetermined communication windows. Further programming complexity is increased due to internal clock drift which sensor nodes are prone to, and therefore clocks must be corrected to accommodate for such drift typically done using time synchronization techniques. The complexity does not end there, the wireless medium over which sensor nodes communicate only allows one node to communicate at a time (within the same transmission range). Therefore,

medium access control (MAC) protocols must be implemented to avoid wireless collisions. Also, sensor networks aim to maximise lifetime by optimising the route taken from sensor nodes to base stations to consume the least amount of energy. A plethora of MAC and routing protocols have been proposed that focus on different aspects including scalability, dynamicity, responsiveness, and density amongst other attributes (Akyildiz et al., 2002; Akkaya and Younis, 2005; Demirkol et al., 2006).

The challenges facing sensor node application programming also stems from the low level embedded programming expertise required. Wireless sensor nodes typically comprise of a microcontroller, a wireless transceiver, a number of sensors and other peripherals such as additional flash storage. Drivers must be implemented for each hardware peripheral which typically communicate over SPI and I^2C, and a strict communication protocol often involving configuring low level registers. Such hardware devices often utilise general purpose input/output (GPIO) pins for status updates which are in turn wired up to microcontroller interrupts. Moreover, substantial internal microcontroller peripherals are wired to interrupts. Thus, knowledge of interrupt based systems and how to program interrupt based systems is required to program such low level embedded systems.

Sensor node applications are predominantly developed in C or flavours of C such as nesC (Gay et al., 2003). Therefore, development challenges also include those faced by traditional low level systems developed in C including allocation and deallocation of memory by the programmer and as well as no type safety mechanisms. The most popular sensor network operating system, TinyOS, allows developers to code applications in nesC (Gay et al., 2003) which exposes an event based programming paradigm. The abstraction layers provided are very low, and as noted by Sugihara and Gupta (2008) "it is often difficult to implement even simple programs." This is partly due to the event based programming paradigm and lack of blocking operations.

As described above, the development learning curve for sensor networks is steep. Higher level languages providing higher abstractions can be used to lower the learning curve substantially. Higher level abstractions can help by hiding the lower level embedded systems and sensor networks specific requirements. Nodes can be put to sleep automatically by underlying drivers; default routing and MAC protocols can be used and swapped without the higher level developer having to change any code in relation to this; drivers for different hardware peripherals can be used and then exposed to the higher level language by abstracting the communication protocols, interrupts and register access. However, in using a higher level language such as Java the developer is also relieved of memory management since this will be taken care of by a garbage collector. Implicit type safety also ensures that the programmer will not incorrectly cast types. More so, majority of the available workforce is already familiar with high level languages such as Java, therefore the learning curve can be drastically decreased be removing the requirement to learn a new language. It has been shown that higher level languages such as Java provide a more efficient development and maintenance environment compared

to lower level languages such as C (Butters, 2007). Therefore, by using a higher level language, the development and maintenance costs of WSNs can be reduced, and thus increase the successful adoption of such technology.

1.1 Enabling Java for Sensor Nodes

Recent work on enabling virtual machines (Brouwers et al., 2009; Caracas et al., 2009; Aslam et al., 2010) for wireless sensor networks attempt to alleviate the paradigm shift by providing a Java programming environment by means of an interpreter. Results presented by Brouwers et al. (2009) demonstrate that an interpreter approach greatly suffers from high execution overheads.

When Java virtual machines (JVMs) for traditional hardware were becoming more popular the costs of interpretation were realised, and initiatives to perform run-time compilation began (Hsieh et al., 1996). However, it is widely assumed that compiling bytecode to native code on such severely resource constrained devices is impossible, infeasible, impractical, complex or resource hungry given the limited storage and memory availability (Palmer, 2004; Koshy and Pandey, 2005; Pandey and Koshy, 2006; Koshy et al., 2008; Aslam, 2011). Following are quotes from recent work which state that run-time compilation is not for WSN class devices:

- Palmer (2004): "...compiled applications, something beyond the power of many of the resources..."
- Koshy and Pandey (2005): "JIT compilers are more practical on higher end platforms."
- Pandey and Koshy (2006): "Most nodes do not have sufficient resources for JIT compilation..."
- Koshy et al. (2008): "JIT compilers are non-trivial programs that cannot be implemented effectively on the typically resource constrained WSN nodes."
- Aslam (2011): "...it is usually difficult to develop a JIT compiler...JIT and AOT are not suitable for a variety of tiny embedded devices..."

Compiling to native code on the development machine would meet the needs to speed up slow interpretation costs. However, drawbacks of this method include that the generated code would be platform specific, and native code tends to be larger than that of bytecode therefore updating nodes over the air would prove to be less efficient. More so, the sensor node would require to perform linking of newly received code since the development machine may not know the location where code will be placed. Therefore, a host machine compilation to native code scheme will not be adequate for updating

sensor nodes remotely. Run-time compilation techniques could be used to achieve both an efficient execution platform as well as provide the possibility of remotely updating sensor nodes.

Therefore, this thesis is concerned with determining whether run-time compilation techniques are in fact something beyond the capabilities of the severely resource constrained devices such as those commonly used in wireless sensor networks (usually having tens of kilobytes of program space and around ten kilobytes of RAM). Techniques to enable both Ahead-Of-Time (AOT) compilation and Just-In-Time (JIT) compilation are proposed, designed, implemented and further compared to existing approaches for enabling a Java execution environment for resource constrained sensor nodes.

Although which programming language would best suit WSN development is an interesting question, it is not the topic of this thesis. Due to the popularity of Java in industry and also recent initiatives to enable Java in WSNs it was decided to use Java as a candidate language. Other high level languages could just as easily be used.

1.2 Challenges, Motivation and Requirements

The main reason why it is assumed that run-time compilation techniques cannot be successfully implemented on wireless sensor network class devices is due to the severe resource constraints they are prone to, being tens of kilobytes of program space and around ten kilobytes of RAM and minimal processing speeds. That said, the limited program and memory availability is the greatest challenge in the face of enabling run-time compilation.

However, perhaps the assumption that run-time compilation is impossible or impractical on such devices is preconceived. The translation logic required to implement a run-time compiler is roughly equal to the translation required to implement an interpreter, thus the program space footprint for the actual translation logic would be similar. The memory required during translation using typical Ahead-Of-Time and Just-In-Time compilation techniques is usually larger than that of an interpreter, however it may be possible to minimize this memory overhead by introducing other techniques. Since the introduction of JIT compilation for Java on traditional platforms, extensive research has been performed in aims of producing highly optimized code by using complex compilation techniques and efficient register mapping. It is perhaps for this reason that run-time compilation techniques may have been deemed unfit for severely resource constrained devices. However, perhaps simple compilation mechanisms could be used which would not require extensive overhead. Results demonstrated from recent interpreter initiatives for wireless sensor networks show the high execution costs of interpreting code. Battery life is another major constraint in wireless sensor nodes, and due to the high

execution overhead of interpreting, battery life may be greatly reduced for computationally intensive (and even for non-computationally intensive) applications. Therefore, the interpreter approach is unsuitable for a large range of WSN applications. Thus, the question that must be asked is whether or not there are other methods to enable high level programming languages such as Java for such resource constrained devices. Chapter 3 further investigates the motivation behind the work and provides a case to continue research in the area of enabling run-time compilation techniques for wireless sensor networks.

The motivation behind this work and one of the primary requirements is to ease the programming burden of wireless sensor networks. However unlike interpreter based approaches previously proposed, execution efficiency is also a primary requirement of this work. The importance of execution efficiency can also directly impact overall system performance as explored in Chapter 3. Development time compilation to native code can achieve this, however does not facilitate over the air reprogramming. Over the air reprogramming is an essential requirement for many wireless sensor networks and therefore reprogrammability also serves as a primary requirement of this work.

1.3 Overview of Work

The work presented in this thesis focuses on run-time compilation techniques for severely resource constrained embedded systems. Run-compilation techniques require other models and components to form a complete system. This work is closely related to bytecode design, language design, application programming interface (API) design, kernel design, driver implementation and programming models, however it is not the scope of this work to explore these areas. The work in this thesis concentrates primarily on enabling run-time compilation techniques for resource constrained devices. In doing so, a case for run-time compilation is made by analysing the effects of battery lifetime for interpreted and native code execution platforms. Ahead-Of-Time compilation is proposed to alleviate slow bytecode execution. In order to achieve a low memory footprint whilst compiling we propose gradual compilation, which compiles code on the fly as it is received. Just-In-Time (JIT) compilation is also proposed for applications that have a larger footprint. In order to achieve JIT compilation we propose using volatile memory for executing code as well as basic block compilation, offline basic block analysis and direct JIT compiler calls. The evaluation of these techniques show that the execution gains achieved are substantial compared with an interpretation approach. More so, the benefits of using run-time compilation is showcased by demonstrating over-the-air reprogramming.

1.4 Relationship to Existing Approaches

Squawk (Shaylor et al., 2003), a Java interpreter based virtual machine developed by Sun Microsystems, was proposed in 2003 for smart cards having 32 bit processors and 160 KB of program space. The virtual machine was later ported to the Sun SPOT (Simon et al., 2006) device which has 512 KB RAM and 4 MB of flash memory. The Squawk virtual machine is intended for devices larger than the devices this work targets. To support the Java virtual machine for lower end platforms than traditional computers, Simon et al. (2006) proposed a Split VM architecture, which allows code to be pre-processed and verified on a more powerful host machine, and then only the necessary code to execute the program is required be sent to the device. In a similar fashion, a Split VM architecture is also adopted in this work, by separating the compilation and verification processes amongst the resource constrained device and the more powerful host machine used to transmit the code.

More recently several initiatives were proposed to enable Java virtual machines for sensor network class devices (Brouwers et al., 2008b; Caracas et al., 2009; Aslam et al., 2008). All three virtual machines propose interpreter based approaches. The standard Java stack (discussed later in Chapter 2) uses a 32 bit width slot. This means that each value put on the stack will take up a multiple of 32 bit slots. Even values consisting of 8 and 16 bits will take up 32 bit slots and therefore waste memory. To overcome this Brouwers (2009) proposes to convert the stack to a 16 bit width stack as originally demonstrated by Lindholm and Yellin (2005). Aslam et al. (2008), on the other hand propose a variable slot size whereas the user can configure the slot size to be 8, 16 or 32 bit. For their test applications, Aslam et al. (2008) show that minimal memory is freed by using an 8 bit slot size, and therefore the authors propose that a 16 bit width stack is sufficient. That said, to support a variable slot the virtual machine footprint is increased, and therefore it is questionable whether this increase justifies the variability of slot size (more so considering that the developer must select the slot size). It is agreeable that a 32 bit stack width will result in a large amount of memory wastage, and since minimal gain is achieved by also offering an 8 bit stack width the approach presented in this thesis proposes using a 16 bit stack width. Two of the virtual machines propose removing textual representations of class, function and field names, since the overhead of keeping such textual representation does not justify their overhead for such memory limited devices. Brouwers et al. (2008b) suggested using a double-ended stack for separation of reference and integer values which would support a more efficient garbage collection scheme and using linked stack management (von Behren et al., 2003) to support threads with less memory wastage. Similar to Brouwers et al. (2008b) the work presented in this thesis also makes use of a double-ended stack and linked stack management. Further details on the techniques employed by the related virtual machines and the work being presented in this thesis are presented in Chapters 2 and 4.

Outside of the realm of resource constrained devices and wireless sensor networks, initiative was put into increasing the slow speed of interpretation. One technique used was Ahead-Of-Time (AOT) compilation, or native code translation (Hsieh et al., 1996). AOT compilation converts bytecode to native code before program execution. AOT approaches are discussed in Chapter 2. In Chapter 4 design and implementation details are given for an AOT compiler for severely resource constrained devices.

Just-In-Time (JIT) compilation is the process of compiling code to native code when it is required to be executed. Java inherently uses a stack to store operands on which all operations are performed. Efficient compilation usually consists of mapping operand operations to registers in an efficient manner. Adl-Tabatabai et al. (1998) and Alpern et al. (1999) propose stack mimicking. This term is used to represent a method by which generated native code performs the exact same operations on the stack as described by the bytecode instructions it is compiling (although such stack operations may not be necessary). In designing a simple compilation process (for both AOT and JIT compilation) it was decided to sustain the underlying Java stack by mimicking the operations which are performed on the stack, except that the operations are performed natively. Due to memory constraints JIT compiling whole functions at a time may not be feasible. Thus, investigation into JIT basic block compilation was undertaken. JIT basic block compilation compiles at the granularity of a basic block. This idea was proposed by Rogers (2002) in aim of increasing JIT execution efficiency. This is due to the use of conditional statements, since JIT compilers may compile code which will not actually be executed. Previous work on JIT compilers is further discussed in Chapter 2 and the approach proposed in Chapter 6.

1.5 Research and Contributions

The scope of this work is to provide grounds as to whether run-time compilation techniques are in fact beyond the resources of devices commonly used in WSNs. The main contributions presented in this thesis are:

- Contrary to the general consensus in the area, it is shown that run-time compilation is in fact possible, practical and can be achieved using simple compilation techniques without consuming large amounts of resources (or at least comparable to existing interpreters).
- The first Ahead-Of-Time and Just-In-Time compilers for such severely resource constrained devices are presented.
- A case demonstrating the benefits of using a native code execution paradigm compared to an interpreted paradigm for wireless sensor networks is provided.

- A technique to reduce the amount of memory required to compile code Ahead-Of-Time is provided, namely Gradual Compilation.
- Although the work is intended for high level development, developers may require lower level register and interrupt access, and therefore a novel approach to exposing such low level concepts to developers was provided.
- An evaluation of both AOT and JIT compilation is provided against an interpreter in which it is shown that both AOT and JIT compilation results in substantially faster execution than that of an interpreter. More so, it is shown that the overhead is comparable to an interpreter.
- Basic block JIT compilation for resource constrained devices is proposed in order to minimize memory overhead requirements when performing JIT compilation.
- Offline basic block analysis is proposed in which basic blocks are identified by a `start basic block` bytecode instruction. This releases the resource constrained device of the task of having to identify the basic blocks itself.
- To minimize the JIT compiler footprint, direct JIT compiler calls are proposed. The stored bytecode for a unit of code will be preceded by a native call to the JIT compiler. Thereafter, by exploiting the underlying hardware architecture, the JIT compiler can establish the location of the bytecode to be compiled and also return directly to the next unit of code to be executed.
- A reprogramming overhead model for interpreted, AOT and JIT compiled code is provided.

The work in this thesis has contributed in part or full to the following publications:

- J. Ellul, K. Martinez (2010). Run-time Compilation of Bytecode in Sensor Networks. In *4th International Conference on Sensor Technologies and Applications 2010.*
- K. Martinez, P. Basford, J. Ellul, R. S. Clarke (2010). Field Deployment of Low Power High Performance Nodes. In *IEEE 3rd International Workshop on Sensor Networks 2010.*
- J. Ellul, K. Martinez. Demo Abstract: Run-time Compilation of Bytecode in Wireless Sensor Networks (2010). In *ACM/IEEE 9th International Conference on Information Processing in Sensor Networks 2010.*
- J. Ellul, K. Martinez (2010). A Few Bytes are Worth a Thousand Words: Run-Time Compilation of High Level Scripts in Sensor Networks. In *IEEE 3rd International Workshop on Sensor Networks 2010.*

- J. Ellul, K. Martinez, D. De Roure (2009). A Dynamic Size Distributed Program Image Cache for Wireless Sensor Networks. In *IEEE 23rd International Conference on Advanced Information Networking and Applications - Workshops 2009.*
- K. Martinez, P. Basford, J. Ellul, R. Spanton (2009). Gumsense - A High Power Low Power Sensor Node. In *6th European Conference on Wireless Sensor Networks 2009.*
- J. Ellul, K. Martinez. DPICache: A Distributed Program Image Cache for Wireless Sensor Networks (2008). In *IEEE/ASME International Conference on Mechtronic and Embedded Systems and Applications 2008.*

1.6 Outline of Thesis

This prelude has provided an overview of the expertise required to program wireless sensor network applications, Java enabling initiatives aimed at lowering the steep learning curve and the inherent execution overheads. The general consensus regarding the impossibility or impracticality of run-time compilation techniques for such resource constrained devices was mentioned and the problems as regards to why this is the consensus in the field.

Chapter 2 serves as an introduction to the field and related work including: wireless sensor networks programming requirements and issues; the typical programming paradigm and other proposed models; the Java programming language, bytecode and supporting virtual machine infrastructure; JVMs proposed for wireless sensor networks are presented along with the techniques they employ; and compilation techniques proposed for traditional JVMs.

Chapter 3 provides a case and motivation to continue research on enabling run-time compilation techniques for such resource constrained devices.

Chapter 4 describes the design requirements and implementation details of how Ahead-Of-Time compilation can be enabled for resource constrained devices. Techniques proposed to enable AOT compilation include a double-ended stack separating reference and integer values, linked stack management, gradual compilation, simple translations which produce a mimic stack and simple optimizations.

Chapter 5 provides an evaluation of the Ahead-Of-Time compiler implementation. It is compared with an interpreter developed for the same specification of devices. Native code is prone to higher encoding sizes. Therefore, thoughts towards JIT compilation are provided.

Chapter 6 demonstrates how Just-In-Time compilation can be achieved in such severely resource constrained devices. To achieve this basic block JIT compilation is proposed,

along with pre-JIT basic block analysis, direct JIT compiler calls and a circular JIT cache.

Chapter 7 evaluates the JIT compilation techniques proposed.

Chapter 8 provides further support for run-time compilation techniques by analysing the reprogramming overheads of AOT, JIT and interpreter based paradigms.

Chapter 9 provides a review of the overall contributions of this thesis and paves the way for future work.

Chapter 2

Related Work

Wireless sensor networks provides a paradigm that is different to traditional computing platforms due to the stringent requirements and various issues inherent in the platform and environment. Higher level languages such as Java can be used to lower the wireless sensor networks learning curve. Previous attempts at enabling such higher level languages have implemented interpretation methods that suffer from high execution overheads. Compilation techniques are typically used to overcome such overheads, however compiling bytecode to native code on development machines is not desirable since it would imply platform dependence, and also larger program updates since native code tends to be larger than bytecode. The general consenus in the community is that on-node compilation is impossible or impractical on such severely resource constrained systems. This thesis is concerned with determining whether this is true or if this notion is preconceived. An introduction to the background and related work will now be provided to allow the reader to appreciate how the work presented in this thesis fits within the broad areas of wireless sensor networks programming, high level languages, bytecode design, virtual machines and compilers.

2.1 WSN Requirements and Issues

Wireless sensor networks consist of a number of sensor nodes that can sense the environment, process the sensed data and transmit (and receive) the processed data for an extended period of time. The requirements and challenges inherent in wireless sensor networks directly influence programmability. Therefore, the main requirements and issues of WSNs will be described here with a focus on how they affect ease of programming.

Wireless sensor nodes are most often equipped with a limited energy source and are usually expected to operate for an extended period of time from months to even years.

A primary requirement of WSNs lies in the fact that they are deployed in environments that are meant to be monitored in an unobtrusive manner. This relies on miniaturisation of nodes and therefore battery size, which in turn requires an energy efficient system. In order to meet the expected lifetime, and given the limited battery source, the system must be as energy-efficient as possible. This involves ensuring that the wireless transceiver, sensors and any other devices are only used and turned on when they are required. This becomes more complex when considering other aspects including MAC protocols, routing protocols and time synchronisation amongst other factors. Programmers are commonly required to explicitly turn different hardware components on and off, and even in to different sleep modes. Such fine grained control of hardware is often intimidating to programmers since they are not familiar with such low level fine grained control. Computational efficiency on the other hand is often considered to be of minor importance and therefore cheaper, more energy-efficient (and slower) processors than those used in larger platforms can be used. However, computational efficiency is often discarded without considering the impact it may have on quality of service and energy expenditure (in Chapter 3 the direct impact of computational efficiency on energy expenditure is demonstrated). Therefore, programming environments should be both as energy and computationally efficient as possible without requiring extensive effort from the system developer.

Networks can consist of a handful of sensor nodes to thousands of sensor nodes. Sensor nodes communicate with each other over the same physical wireless medium. Therefore, protocols and overall system implementation must be able to scale with the network size and limited bandwidth. Developers typically have to cater for such low level intrinsic properties when they really should be able to concentrate effort on application requirements. Besides the work involved in deploying a WSN, one must also keep in mind that like other computing platforms, sensor nodes may be required to be updated from time to time due to various reasons including bug fixes, new application requirements and even complete retasking of a network. Therefore, software reconfiguration and reprogrammability is essential, however this is often left up to the developer to implement.

The underlying theme from the above is that the low level internals of WSNs is more often than not left up to application developers to implement, when really they should be focusing on the application specific requirements. The work in this thesis focuses on easing the programming burden by providing a higher level sensor node programming abstraction. Other programming models attempt to achieve this with higher level abstractions. A taxonomy of the different programming models proposed will now be discussed.

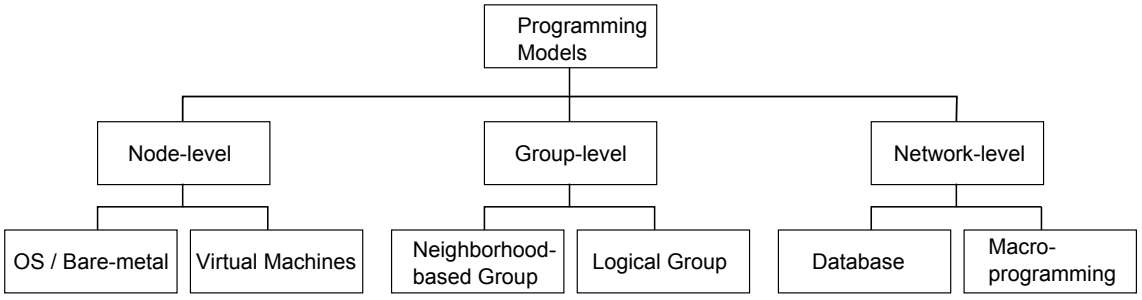

Figure 2.1: A taxonomy of sensor networks programming models.

2.2 WSN Programming Paradigms

Programming applications for wireless sensor networks can be a daunting task due to the low level embedded systems platform and the stringent requirements inherent in WSNs. This had been identified as a problem since the introduction of WSN technology and over the past decade different approaches have been proposed to overcome the steep learning curve. The different programming models proposed can be divided into Node-level, Group-level and Network-level abstractions. Figure 2.1 reproduces the taxonomy of programming models as proposed by Sugihara and Gupta (2008).

2.2.1 Node-Level Programming

Node-level programming environments provide the lowest level abstraction in that hardware is abstracted such that programmers can control the underlying devices as required which provides the greatest flexibility. Program logic consists of controlling each node individually and describing how individual nodes sense the environment, process sensor data and interact with neighboring nodes. Initial WSN deployments were typically programmed using C on the bare metal (i.e. code that executes directly on the microcontroller and requires to interface with the different hardware peripherals) by embedded systems developers. Due to various reasons including the growing interest in wireless sensor network applications, effort was put into allowing non embedded experts to develop their own applications. Such effort led to operating systems development and soon after virtual machines development. An extensive number of approaches have been proposed in the past decade, therefore only the most popular and related approaches will be described here.

2.2.1.1 OS / Bare-metal Programming

Bare-metal programming refers to programming that involves directly interfacing with the underlying hardware including the processor's registers, peripherals and other hardware. Programming 'on the bare-metal' (as it is referred to) is commonly developed in

C and/or Assembly (for highly optimised code sections). Due to the low level required to program 'on the bare metal', initiative was put into abstracting the hardware layer so that non-embedded programmers would also be able to develop applications for WSNs. TinyOS (Levis et al., 2004) is one of the earliest attempts to abstract bare-metal programming and is also referred to as the *de facto* standard operating system. TinyOS programs are developed using nesC (Gay et al., 2003), a language based on C that provides component abstractions and is event based. Different system components are 'wired' together in configuration files to form a single application. Components and the wiring concept were most likely proposed to encourage code reuse and portability of code across different hardware, however reusing code across different hardware often requires digging deep into low levels of the TinyOS libraries. Although, TinyOS is widely used it is also widely accepted that it is extremely difficult to use and even configure to different application requirements. This may be due to poor application programming interface (API) design, the event driven programming paradigm or perhaps to the fact that a new language must be adopted. Applications coded in nesC are actually compiled to C and merged with the underlying TinyOS components that are used. Thereafter the generated C code is compiled to a single native code binary and is loaded directly onto the bare-metal. The term operating system in the WSN community is somewhat different to traditional computing. Operating systems tend to include elements of a separate execution kernel that provides services to applications running on top of it (Silberschatz et al., 2001). TinyOS does not provide a kernel but is essentially a framework or library of existing components. Many other 'operating systems' proposed for WSNs use similar approaches. The main problem with single native binary approaches (that consist of both the operating system and the application) is that reprogramming is either not possible or else can be implemented without the option of reusing code that is already installed on the system.

Contiki (Dunkels et al., 2004) is another popular operating system used for WSNs. Programs are coded in C and both event-driven and threaded programing models can be used. Light-weight pre-emptive threading can be implemented using Protothreads (Dunkels et al., 2006b) which only requires two bytes of memory per protothread. Contiki also supports dynamic program loading which allows for top level application logic to be replaced as necessary. Contiki provides an extensive set of libraries and drivers that can easily be used, however like many other operating system approaches in the WSN community does not address supporting higher level programming languages for non embedded programmers.

2.2.1.2 Virtual Machines

Virtual machines abstract underlying hardware by providing a higher level execution platform. Virtual machines can provide different benefits including higher level programming languages and APIs, smaller sized program encodings (which is useful for reprogramming WSNs) and platform independence. However, the following disadvantages are inherent in supporting a virtual machine: increased program space and memory requirements and any execution overheads incurred due to the virtual machine abstraction. Virtual machines can execute code by either interpreting the code or else by compiling the code to the underlying hardware's native code. Interpretation and compilation is further discussed later in this chapter.

Maté (Levis and Culler, 2002) is one of the first virtual machines proposed for WSNs. Maté's main goal was to provide an energy-efficient means of retasking deployed nodes. Maté executes virtual machine scripts termed Maté scripts that are encoded in 'capsules'. The scripts are comprised of assembly like instructions that can perform arithmetic and boolean operations, manipulate the stack, high level hardware configuration (e.g. turn sounder on/off), and calling user functions. The instruction set can be tailored for different applications, and therefore the approach was termed as 'Application Specific Virtual Machines' (ASVMs) (Levis et al., 2005). By exposing such a high level instruction set the encoding size can be drastically reduced, and as a result of this the flexibility exposed through the language is also drastically limited. The VM is implemented as an interpreter and therefore high execution overheads are inherited (over 30% overhead for arithmetic operations). More recent Java based virtual machine approaches are described later in this chapter.

2.2.2 Group-Level Programming

Group-level models abstract the individual nodes into groups of nodes, whereby application logic can be defined on a group level rather than treat each node as an individual system. This approach can be considered to be a system of systems. The higher abstraction allows for the developer to focus more on the overall system application, however at the cost of less flexibility. The group-level model is split into neighbourhood-based grouping which relies on spatial locality whilst logical grouping tends to encapsulate those programming models that group nodes together based on other logical properties, such as by the type of sensors.

Neighbourhood-based grouping abstracts the wireless sensor network into groups of sensors according to their spatial locality. Each sensor node defines its own group locally by establishing its neighbouring nodes. Spatial locality is a good fit for many WSN applications since readings are often highly correlated to the location of sensor nodes. Spatial locality is also inherent in the transmission range of the wireless medium used in WSNs.

Abstract Regions (Welsh and Mainland, 2004) and Hood (Whitehouse et al., 2004) are two popular neighbourhood-based programming abstractions proposed for WSNs. The languages expose automatic neighbourhood discovery and data sharing abstractions to facilitate group based processing of data.

Spatial locality may not be the ideal grouping for sensor nodes in regards to processing data for certain applications. Therefore, other logical grouping abstractions were proposed whereby groups are defined by other properties. Such properties can include (but are not limited to) types of sensors available, other environmental parameters and even based on the dynamic input sensed from the environment. More so, group membership can be dynamic allowing nodes to join and leave groups according to the group membership criteria. A particular example where spatial locality is not the ideal programming abstraction is that of target-tracking based applications. Abdelzaher et al. (2004) proposed EnviroTrack, a grouping abstraction focused on target tracking. Sensor nodes are grouped to all the sensor nodes that detect the same event. A generic logical grouping programming abstraction language, SPIDEY, was proposed by Mottola and Picco (2006). Nodes are assigned attributes and group membership can be defined by predicate logic based on the assigned attributes. Thereby the developer can specify the logic which defines a group. Such groupings can consist of all nodes with the same sensor type, or even all nodes with specific sensor input ranges (for example all sensor nodes having temperature readings greater than 10 degrees).

2.2.3 Network-Level Programming

The network-level programming model provides the highest level of abstraction by abstracting the whole network as a single abstract machine. Again, the higher abstraction will provide an easier programming framework however again at the cost of further restrictions on flexibility.

The database programming abstraction was the first network-level programming model proposed for WSNs. Primary work on WSN database abstractions includes Cougar (Bonnet et al., 2000), SINA (Srisathapornphat et al., 2000) and TinyDB (Madden et al., 2005). The approaches allow for sensor data to be queried using SQL-like queries via a single query engine abstraction. The approaches vary in how the retrieval of sensor data is implemented. Cougar was the first proposed database programming approach in which the SQL-like queries are disseminated into the wireless sensor network. Each sensor node will upon receiving the query, process the query by sampling sensors and performing any specified aggregation of previous stored data and transmit the results back to the query originator. In addition to this, SINA also provides an imperative programming language called Sensor Querying and Tasking Language (SQTL) that allows for tasks to be written much like that of stored procedures in traditional databases. TinyDB extends the query request paradigm by also adding constructs to provide automatic

query updates by specifying how often data should be sampled, how often the processed query results should be transmit and also on which nodes the tasks should take place. Work on routing trees to provide optimal paths for disseminating queries and collecting results was also proposed. The database programming abstraction provides an easy to user interface to query readings, perform simple aggregations on data, and transmit the data to the base station. The trade-off is however that more sophisticated tasks cannot be implemented.

Macroprogramming languages were proposed that attempt to provide ease of programming without trading off extensive flexibility. Regiment (Newton and Welsh, 2004) is one of the earliest proposed macroprogramming languages for WSNs. Regiment is implemented as a functional programming language similar to Haskell (Hudak et al., 1992). Groups of nodes can be specified according to spatial and topological location as well as other logical parameters, and thereafter operations can be performed on the groups. Operations can be performed on individual nodes or groups of nodes and their results aggregated to provide a single system wide view of the collected and processed data.

Recent work by Hossain et al. (2011) acknowledge the benefits heeded using the higher level macroprogramming abstraction and propose that node-level microprogramming can be made to be just as simple as macroprogramming given an adequate programming abstraction. To demonstrate this Hossain et al. (2011) propose μSETL, a programming abstraction that allows node-level programming using set theory like operations. It is shown that by using set theory based operations with the 'right set of programming abstractions' programs can be described with comparable lines of code to that of other macroprogramming approaches for several relatively simple applications. It is not shown whether or not the approach will be adequate for more complex applications. More so, as noted by the authors the ease of programming (or number of lines of code in a program) really depends on the abstractions available. More complex applications requiring abstractions that are not provided by the run-time environment will result in more effort from the programmer. Really, as indicated by Hossain et al. (2011), it is the right set of programming abstractions that ease the programming complexity and not the language. As previously stated, although which programming language and programming abstraction best suits WSNs is an interesting topic, it is not the topic of this thesis. Java was chosen as a candidate language in this thesis due to its popularity and also since other virtual machines proposed used Java, therefore a comparison between the different approaches would be more fair. Background on the Java programming language, bytecode and virtual machine specification will now be provided.

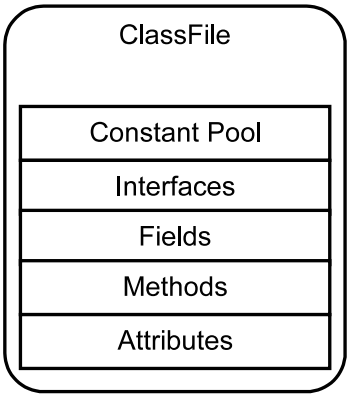

Figure 2.2: An overview of the Java `ClassFile` structure.

2.3 Java

The Java programming language (Gosling et al., 1996) is a general-purpose object-oriented language designed with simplicity in mind. Its syntax is similar to C and C++ with a few alterations. Platform independence is another focal design criteria of the Java framework. The authors state that code should be able to be written without an understanding of the underlying hardware architecture. This would allow programmers with minimal low level expertise to easily develop solutions for ranges of different platforms. Java over the years has matured into a popular and industry accepted language used in a plethora of environments.

The Java Virtual Machine (JVM) as specified by Lindholm and Yellin (1999) describes the internals of JVMs and how they should be constructed. Here a description will be provided of the concepts and principles which are necessary for comprehension of the work presented in this thesis in relation to Java. This section is not intended as an introduction to the language constructs and uses, but to provide an overview of the underlying framework that supports the virtual machine and execution paradigm.

2.3.1 The Class File Format

Java code is written and separated into different Java class source files. Java source is then compiled using the Java compiler which produces Java `ClassFile` structures. The `ClassFile` structure's overview is presented in Figure 2.2. The `ClassFile` structure's main components include the constant pool, interfaces, fields, methods and attributes. The constant pool contains class, interface, field and method names (where class and interface names are stored as fully qualified Java names) as well as strings and other constant values. A list of interfaces is stored containing all interfaces that act as super-interfaces to the class. All of the class' fields and methods are listed in the `ClassFile` structure and the associated bytecode for each method is stored within the method

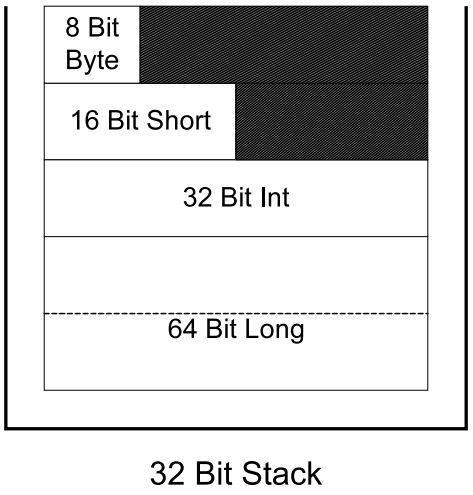

Figure 2.3: A 32 bit operand stack depicting usage of 8, 16, 32 and 64 bit datatypes.

structure. Attributes are also stored in the `ClassFile`. Attributes allow for class meta-information to be associated to the class itself as well as to specific fields and methods. The Java framework specifies that class files should be verified to ensure that the code within the class file is valid and respects the framework's constraints.

2.3.2 JVM Stacks, Method Frames and Operand Stacks

A JVM stack is used to store frames, where a frame contains data related to a specific method. A JVM stack is created for each thread of execution within the JVM. When a new method is called, a new frame is created and associated to the new method, and pushed onto the JVM stack. When a method is finished executing, its associated frame will be popped from the JVM stack. Thereby only one frame will be active at any point in time (in each thread of execution). A method's frame will consist of local variables, the method's operand stack and other method related information such as a return address and other JVM implementation specific information. As mentioned each frame will contain an operand stack which is used to store operands and results from the operations specified in associated method bytecode. The Java bytecode instructions perform operations (to and from the stack) at the smallest granularity of 32 bits. Figure 2.3 depicts a (Java) 32 bit operand stack containing an 8, 16, 32 and 64 bit values stored in it. A 64 bit value will take up two slots on the stack, a 32 bit value takes up a single slot, while a 16 bit and 8 bit value will also take up a single slot. That means that 16 bits of memory will be wasted when a 16 bit value is placed on the stack and 24 bits is wasted when an 8 bit value is placed on the stack. Since most operations on traditional platforms consist of 32 bits, and given the abundance of memory resources, this is hardly an issue when targeting traditional platforms.

2.3.3 Java Bytecode Instruction Specification

An overview of the Java bytecode specification will be described here, however for a more comprehensive description interested readers should refer to the actual specification by Lindholm and Yellin (1999).

After Java source files are compiled, Java `ClassFile`s are produced which contain method structures, whereby each of these method structures will contain bytecode which represents the actual method logic. Bytecode instructions represent operations to be performed. When necessary instructions are required to pop values from the operand stack to use inputs to the operation and also may push results on to the stack. Operations may also result in storing and loading values from and to variables or fields. The standard notation for describing an operation's effect on the stack is as follows:

```
..., value1, value2 ⇒ ..., value3
```

The left side of the right arrow represents the stack before the operation takes place and the right side of the right arrow represents the stack after the operation has taken place. The three dots ('...') represent the stack and its contents which are not relevant to the respective operation. All inputs to the operation are then specified (in the case above value1 and value2). The above then describes that value1 and value2 are popped from the operand stack, and thereafter value3 is pushed onto the operand stack. The example above would suit the stack effects for an `iadd` bytecode instruction. The `iadd` instruction is used to add two integer values. In doing so it will pop two values from the stack, value1 and value2, perform an addition of the two and the push the result, value3 on the stack (where value3 = value1 + value2).

Bytecode operations consist of single byte opcodes that specify the operation which the bytecode operation performs, along with zero or more operands. A single byte opcode implies that there can be a maximum of 256 different operations, however of the possible 256 possibilities 51 are reserved for other purposes or future use. The number of operands that each bytecode instruction requires as specified in the Java Virtual Machine Specification (Lindholm and Yellin, 1999) is dependent on the opcode itself. The opcode itself also determines the type of the parameters it requires. For example the `iload` instruction loads a local `int` variable, whilst the `fload` instruction loads a `float` variable onto the stack. As per the examples, most of the typed bytecode instructions can be identified by the first character of the instruction where `i` represents `int` operations, `l` represents `long` operations, `s` for `short` operations, `byte` operations by `b`, `c` for `char` operations, `f` represents `float` operations, `double` operations by `d` and `a` is used to represent `reference` operations. However, a bytecode operation does not exist for every possible typed operation since this would require more than the 256 possible bytecode opcodes, also since the Java stack is inherently 32 bits (i.e. every value

Table 2.1: Typed Java Bytecode Instructions

opcode	byte	short	int	long	float	double	char	reference
Tipush	bipush	sipush						
Tconst			iconst	lconst	fconst	dconst		aconst
Tload			iload	lload	fload	dload		aload
Tstore			istore	lstore	fstore	dstore		astore
Tinc			iinc					
Taload	baload	saload	iaload	laload	faload	daload	caload	aaload
Tastore	bastore	sastore	iastore	lastore	fastore	dastore	castore	aastore
Tadd			iadd	ladd	fadd	dadd		
Tsub			isub	lsub	fsub	dsub		
Tmul			imul	lmul	fmul	dmul		
Tdiv			idiv	ldiv	fdiv	ddiv		
Trem			irem	lrem	frem	drem		
Tneg			ineg	lneg	fneg	dneg		
Tshl			ishl	lshl				
Tshr			ishr	lshr				
Tushr			iushr	lushr				
Tand			iand	land				
Tor			ior	lor				
Txor			ixor	lxor				
i2T	i2b	i2s		i2l	i2f	i2d		
l2T			l2i		l2f	l2d		
f2T			f2i	f2l		f2d		
d2T			d2i	d2l	d2f			
Tcmp				lcmp				
Tcmpl					fcmpl	dcmpl		
Tcmpg					fcmpg	dcmpg		
if_TcmpOP			if_icmpOP					if_acmpOP
Treturn			ireturn	lreturn	freturn	dreturn		areturn

that lives on the stack takes up 32 bits) than stack operations that involve pushing or popping values less than 32 bits would never be used. Table 2.1 provides a summary of the type support provided by the bytecode instruction set as reproduced from Lindholm and Yellin (1999). If an operation is not provided for a specific value type, then the value can be converted to a type that is supported for the specific bytecode operation. For example, if an addition is required for two `short` values, then the values can be converted to int values using the short-to-int, `s2i`, bytecode instruction. Table 2.1 is summarised by each operation in the opcode column where the capital T represents the parameter type, and in the case of `if_TcmpOP`, OP represents the operation type. A description of each bytecode operation group including bytecode instructions that are not summarised by parameter types in Table 2.1 will follow.

2.3.3.1 Load and Store Instructions

JVMs use a stack to store input values for operations and the results of operations are also placed on the stack. Values will however not live permanently on the stack. Values are typically stored in local variables that live inside a JVM method frame as described above. To transfer local variables from JVM method frames to the operand stack the following load instructions are used: `iload`, `lload`, `fload`, `dload` and `aload` to transfer `int`, `long`, `float`, `double` and `reference` values respectively. These load bytecode operations require a single byte parameter to indicate the index number of the local variable

to load onto the stack. Since loading values from local variables to the stack is a very common operation the JVM bytecode specification implements bytecode mnemonics of the load operation that does not require a separate local variable index parameter in the format of: `iload_<n>`, `lload_<n>`, `fload_<n>`, `dload_<n>` and `aload_<n>` where `<n>` is replaced for each local variable index from 0 to 3. Similarly for storing values stored on the operand stack back into local variables (stored in the JVM method frame) the following bytecode instructions can be used: `istore`, `istore_<n>`, `lstore`, `lstore_<n>`, `fstore`, `fstore_<n>`, `dstore`, `dstore_<n>`, `astore` and `astore_<n>`.

Java needs to facilitate more than 256 local variables, therefore the `wide` bytecode instruction allows for the execution of the load and store instructions with a 2 byte local variable index.

Constant values can be pushed onto the operand stack using `bipush` and `sipush` which push `byte` and `short` values as `int` values respectively; `ldc`, `ldc_w` and `ldc2_w` are used to push constants stored in the constant pool onto the operand stack; `aconst_null` pushes the `null` reference value; `iconst_m1` pushes -1 onto the stack; and `iconst_<i>`, `lconst_<l>`, `fconst_<f>` and `dconst_<d>` are used to store constant values onto the stack where `<i>`, `<l>`, `<f>` and `<d>` represent 0 to 5, 0 to 1, 0 to 2 and 0 to 1 respectively.

2.3.3.2 Arithmetic Instructions

Arithmetic instructions supported can be grouped into single operand and dual operand instruction groupings. Dual operand instructions pop the two operands from the stack, perform the arithmetic operation and push the result onto the stack. The dual operand instructions support addition (`iadd`, `ladd`, `fadd` and `dadd`), subtraction (`isub`, `lsub`, `fsub` and `dsub`, multiplication (`imul`, `lmul`, `fmul` and `dmul`), division (`idiv`, `ldiv`, `fdiv` and `ddiv`), remainder (`irem`, `lrem`, `frem` and `drem`), shifts (`ishl`, `ishr`, `iushr`, `lshl`, `lshr` and `lushr`), bitwise or (`ior` and `lor`), bitwise and (`iand` and `land`), bitwise xor (`ixor` and `lxor`), comparisons (`dcmpg`, `dcmpl`, `fcmpg`, `fcmpl` and `lcmp`) and incrementation, `iinc`, which differs from addition in that it increments a local variable by a specified value.

Single operand instructions similarly pop a single operand, perform the operation and push the result on the stack. The only single operand arithmetic operation supported is negation (`ineg`, `lneg`, `fneg` and `dneg`).

2.3.3.3 Type Conversion Instructions

Type conversion instructions are used to convert a value from one type to another. This is typically required either due to conversions explicit in user code or else due to conversions required to perform typed operations that are not supported. Conversions can

be grouped into two types, widening conversions and narrowing conversions. Widening conversions are those conversions that can convert a value to a data type that supports all possible values of the original data type (or more). Widening conversions supported include `int` to `long` (`i2l`), `int` to `float` (`i2f`), `int` to `double` (`i2d`), `long` to `float` (`l2f`), `long` to `double` (`l2d`) and `float` to `double` (`f2d`).

Narrowing conversions changes values to a data type that cannot hold all possible values of the original data type. The narrowing conversions supported are `int` to `byte` (`i2b`), `int` to `char` (`i2c`), `int` to `short` (`i2s`), `long` to `int` (`l2i`), `float` to `int` (`f2i`), `float` to `long` (`f2l`), `double` to `int` (`d2i`), `double` to `long` (`d2l`) and `double` to `float` (`d2f`).

2.3.3.4 Object Creation and Manipulation

Instances of objects are created using the `new` bytecode instruction. The type of object is specified by two bytes placed on the stack which consist of a constant pool index that points to a class reference type. Arrays are created using the `newarray`, `anewarray` and `multianewarray` which are used for standard value type arrays, reference arrays and multi-dimensional reference arrays.

Values of object fields and class variables (or static fields) are retrieved using `getfield` and `getstatic` respectively. The constant pool index associated with the field or class variable is included as an operand following the opcode. In the case of an object field the object reference is placed on the stack prior to execution of the bytecode instruction. Thereby the virtual machine can look up the object reference in the case of object fields, and for both opcodes, the virtual machine will place the object field's value or the class variable's value on the stack. Storing values works in a similar fashion. The object field or class variable constant pool index is specified as an operand to the `putfield` and `putstatic` instructions respectively. The value to be stored is expected to be on the stack prior to execution of the bytecode instruction. In the case of object field value storing the object reference is also expected to be on the stack.

Array values can be retrieved by looking up the array reference and then finding the required array item to be retrieved. Both the array reference and array item index are expected to be on the stack. The instruction will then pop the array reference and the array item index, find the array item to be retrieved and push the array item's value on the stack. To store values into an array, the value to be stored along with the array reference and array item index is expected to be on the stack. Array item value loading and storing is supported for the standard Java value types: `byte` (`baload` and `bastore`), `char` (`caload` and `castore`), `short` (`saload` and `sastore`), `int` (`iaload` and `iastore`), `long` (`laload` and `lastore`), `float` (`faload` and `fastore`) and `double` (`daload` and `dastore`) and reference arrays (`aaload` and `aastore`).

Other object and array bytecode instructions include instructions to check the type of an object or array (`instanceof` and `checkcast`) and array length retrieval (`arraylength`).

2.3.3.5 Control Transfer Instructions

Execution flow can be altered using control transfer instructions. Unconditional branching supports instructions that can change execution to a code address specified as a parameter both without the intention of returning to the original execution point (`goto` and `goto_w`) as well as instructions that support returning to the original execution point (`jsr` and `jsr_w`) by placing the return address on the stack. Unconditional jump locations can also be specified by the value of local variables (`ret`).

Simple conditional instructions are also used to alter execution flow. The Java bytecode specification supports instructions that alter control flow based on: the comparison of a value (on the stack) with the integer 0 (`ifeq`, `iflt`, `ifle`, `ifne`, `ifgt` and `ifge`); the comparison of a reference (on the stack) with the null reference (`ifnull` and `ifnonnull`); the comparison of two integer values on the stack (`if_icmpeq`, `if_icmpne`, `if_icmplt`, `if_icmpgt`, `if_icmple` and `if_icmpge`); and the comparison of two references on the stack (`if_acmpeq` and `if_acmpne`). Each simple conditional instruction requires a two byte parameter that represents a signed offset to jump to in the case of when a condition is evaluated to be true.

Compound conditional instructions are also supported that allow for multiple branching destinations based on the value of an integer (on the stack). The `lookupswitch` bytecode instruction requires that each comparison value-destination pair follows the opcode. A comparison value-destination pair consists of a comparison value by which the value being compared is matched against. If the value (on the stack) matches the comparison value then execution is branched to the address specified in the comparison value-destination pair. It is often the case that the individual comparison values are provided in a sequential order (when using `switch-case` statements). Therefore, the `tableswitch` instruction was also provided that does not require comparison values but only a list of destination addresses. The minimum comparison value and maximum comparison value are passed as parameters to the instruction. Thereby, the JVM can compare each value in the range and jump to the relative destination if a match is found.

2.3.3.6 Method Invocation and Return Instructions

Four method invocation instructions are provided by the bytecode specification including: `invokevirtual`, the standard method of dispatching method calls; `invokeinterface`, used to invoke a method according to the interface method specification; `invokespecial`, used for invoking instance initialisation methods, private methods or a superclass method;

and `invokestatic`, which is used to call class methods (attributed by the `static` keyword).

Method control flow starts at the beginning of a method and ends when a return statement is reached. A return statement must always end execution of a method (even if it is the last instruction in a method). Methods that do not return values use the `return` instruction to identify that execution of the method has ended. If a method returns a value then the associated return instruction is used. `ireturn` for `int`, `lreturn` for `long`, `freturn` for `float`, `dreturn` for `double` and `areturn` for references.

2.3.3.7 Exceptions

Java supports the throwing and catching of exceptions. The only bytecode instruction implemented to support this is the `athrow` instruction which instructs the JVM to throw an exception where the exception object is expected to be on the stack. The process of finding an exception handler and passing uncaught exceptions up the call stack is left up to the JVM.

2.3.3.8 Synchronization Instructions

Synchronization is implemented using two instructions `monitorenter` and `monitorexit`. The `monitorenter` instruction is used to acquire a *lock* on a specified object (which is expected to be on the stack). An object's lock can only be acquired by one thread at a time. The `monitorexit` instruction is used to release an acquired lock of a specified object (which is expected to be on the stack).

2.3.3.9 Operand Stack Management Instructions

Direct manipulation of the data on the stack can be implemented using instructions which: pop items from the stack (`pop` and `pop2`); duplication of items on the stack (`dup`, `dup2`, `dup_x1`, `dup2_x1`, `dup_x2` and `dup2_x2`); and swap items on the stack (`swap`).

2.4 Desirability of Java for WSNs

The above provides an overview on the Java Virtual Machine specification and how Java bytecode is constructed. Given the deeper insight into the internals of the virtual machine and language it is sufficient to now analyse the desirability (and undesirability) of Java for wireless sensor networks. The main benefit heeded using Java as a programming language lies in the fact that most computer science students and workforce are proficient

in Java programming or in the very least familiar with it. Thereby, the learning curve to develop WSN applications would greatly be minimised. More so, as previously noted the development and maintenance costs of systems implemented in high level languages such as Java are reduced than lower level languages such as C (Butters, 2007). Previous work proposing popular high level languages have focused on Java as a programming language (which will be described next). Therefore, it is suitable to implement Java as the language of choice in order to not introduce any unfair comparisons (that said, the run-time compilation techniques proposed in this thesis are not specifically tied to Java and can be used with other languages).

The high level programming abstraction provided by Java, along with an adequate hardware abstraction can greatly decrease programming effort. However this is a double-edged sword. Since like many of the advantages introduced by abstracting to higher programming levels, undesirable features are also introduced including the execution overhead introduced due to the abstraction. This is a trade-off that must be evaluated for such programming paradigms. That said, the work proposed in this thesis attempts to minimise such execution overheads without sacrificing programmability. Another trade-off implicit in abstraction is the loss of fine-grained control of underlying hardware, however that is essentially the goal of the higher level abstraction. More so, such fine-grained tuning can be implemented at the operating system or driver layer if required.

Java programs are stored and distributed as Java bytecode. The benefits of this include platform independence, and also bytecode tends to be smaller in size than that of native code which are both desirable features for over-the-air reprogramming. Java bytecode however was not designed with 8 or 16 bit microcontrollers in mind. This can be seen from table 2.1 whereby `byte` and `short` data types (that is 8 and 16 bit data types) are not inherently supported for most bytecode operations. Most operations on `byte` and `short` datatypes require casting to or from the `int` (32 bit) datatype. This means that more than double the memory is required on the stack for 8 or 16 bit operations. Also, execution is slowed down due to the introduced typecasting operations. For this reason an intermediate assembly language that can inherently support operations of 8 or 16 bits would be better suited such as that of the Common Language Infrastructure (CLI) (Miller and Ragsdale, 2003) used in .NET or the LLVM assembly language (Lattner and Adve, 2004).

That said, the desirability or undesirability of the usage of the Java programming language for WSNs is subject to the more generic question as regards to whether node-level programming is a suitable feature for WSNs or if higher level group or network level abstractions are more suited. This question is out of scope of this thesis, however it should be noted that a majority of WSN applications are developed using a node-level programming model. Therefore, the work in this thesis provides methods for implementing a node-level programming environment which is easier to use than alternatives without sacrificing execution efficiency. Also, the premise that an extensive number of

computer science graduates and available workforce are already knowledgeable in Java programming, may provide an advantage over learning new group-level and network-level programming models. However, again this is out of scope of this thesis and should be addressed in future work.

As mentioned before to minimise unfair advantages, we have decided to use Java in a similar manner to other approaches proposed for WSNs. To overcome this problem previous approaches propose analysing the bytecode and altering it to inherently support 8 or 16 bit operations as will be described further below.

2.5 Sensor Nodes are for WSNs, not for Java Purists

Java provides a number of different *Java Editions* aimed at providing different needs to different device classes. The most notable are Enterprise, Standard and Micro Editions which are targeted at enterprise software, generic pc based applications, and embedded systems (larger than that of WSNs) including mobile phones and set-top boxes.

A less popular platform more targeted towards the smaller sizes of WSNs (or even smaller) was proposed for smart cards, namely JavaCard. The JavaCard platform however is too specific to smart cards and not generic enough for WSNs. Restrictions include lack of `float` and `double` datatypes and therefore does not support any decimal values which may be required by certain developers. JavaCard programs are implemented in a web server request and response paradigm in Java `Servlet`s which is not the ideal execution paradigm for WSNs, also the implementation requires extensive security features that are not necessarily required in WSN applications.

2.5.1 Need for a New Java Platform Specification

Two predominant schools of thought exist regarding how Java should be supported on sensor nodes, the first being that of Java purism and the second group, focusing on sensor node optimization which will be named pragmatics here for lack of a better word. Java purists believe that the port should be compliant with a Java Edition (usually the Java Micro Edition since the JavaCard platform is not well suited). These implications result in an increase of cost or substantial decrease in memory availability and most likely higher execution overheads. Pragmatics, on the other hand, focus on the task at hand and how it can best be achieved; i.e. how can Java be supported (or a subset thereof) which will allow (novice) programmers to easily write applications for sensor networks without sacrificing memory or increasing execution overhead or node costs. The Java Micro Edition supports two configurations: the Connected Device Configuration (CDC) and the Connected Limited Device Configuration (CLDC). The CLDC defines the lowest common denominator of features amongst the different editions. It is the opinion of the

author of this thesis that the CLDC specification is not adequate for WSNs and a new specification should be devised. Reasons behind this opinion are described here.

The CLDC specification was primarily intended for larger devices than the sensor nodes commonly used today, and thus the design of CLDC was not focused around sensor nodes. Implementing a CLDC compliant sensor node would most likely increase costs due to requiring more memory, or substantially limit the amount of memory available to the developer's application. This is ultimately for features which are useful only to a very few number of sensor applications. Reflection allows an application to, at run-time, inspect and modify applications in the VM. How useful is that to sensor network applications, and is it worth the overhead trade-offs? CLDC does not support reflection and more so reflection can provide little benefit for WSNs (especially compared to the trade-offs required to support it). CLDC does limit the Java platform quite suitably, however also requires certain features that are not required in WSNs which will be highlighted here.

2.5.1.1 Strings and Encodings

CLDC includes a number of classes which provide text related features such as strings, character encodings and related functions. Sensor nodes do not require to output data in a textual representation, nor do they require to input data in such an encoding. Thus, the related classes can be omitted for a sensor node implementation. Strings may be useful for debugging purposes, however they provide no real benefit for executing an application.

2.5.1.2 The `double` Datatype

MSPGCC and AVR-GCC, the most common MSP430 and AVR compilers used in sensor networks, do not support the double (64 bit) floating point datatype. Although this fact is not justification to omit the datatype it should be stated that many sensor network deployments have managed fine without a 64 bit floating point implementation. Although, scenarios may exist where such precision is required the overhead required for supporting a double datatype should not be inflicted on all sensor network applications.

2.5.1.3 IO, Streams and Connections

Streams, StreamReaders and Writers are a concept which most Java developers should be familiar with. They provide an abstraction to interact with a stream of bytes usually associated with files, network connections or memory. The question must be asked as to whether this is an appropriate abstraction for sensor nodes. Sensor nodes communicate

with attached sensors and peripherals with analog to digital circuitry as well as data buses including serial, I^2C and SPI amongst others. Communicating with a sensor often includes writing 'tight' code which configures the sensor if required, and then reads raw or preprocessed sensor values. Sensor communication often involves strict timing requirements as well as initializing the sensor which may include warming up the sensor. Also, sensor values are usually read and then either stored, sent over the air or have some computation performed on them. The concept of applying a finite stream of data does not fit here.

Applying streams to communication mediums only fits slightly better. The CLDC specification includes the Generic Connection Framework which provides an abstraction for connection oriented communication. The general form of opening a communication connection is described as follows: `Connector.open("<protocol>:<address>;<parameters>")` A String representation of a connection is most likely not the optimal way to describe a sensor network connection. Also, sensor network communication protocols are often coupled tightly with the application. The API in the CLDC specification does not cater for such coupling.

2.5.1.4 Calendars and Dates

Many sensor networks only require knowledge of a global sensor network time, and not any relation to time outside of the sensor network. Other sensor networks may require information to do with the time external to the sensor network, and thus parts of the `Calendar, Date` and `TimeZone` classes may be useful. However, it is unfair to impose the memory requirement for such classes on sensor nodes which would not require such functionality.

2.5.1.5 Dynamic Class Loading

The CLDC specification requires that dynamic class loading is implemented. This is a feature which is rarely required in desktop applications, let alone in sensor networks which usually have a single purpose and are maintained by a single developer or team.

As with most aspects of sensor networks it is a running theme that the inclusion of such features really is application specific. Thus, it would be beneficial that each sensor node application (or deployment) can be configured to include the system classes which it requires.

2.5.1.6 Exceptions and Threads

Other Java features that may or may not be required for WSN applications include Exceptions and Threads. Both involve an additional memory, program space and execution overhead. Some may argue that exceptions can be removed by replacing them with function return values that signify whether an error or unexpected state has occurred. On the other hand perhaps exceptions are a programming notation that developers require. Similarly the requirement of threads is just as controversial. Some argue that threads are essential whilst others argue that event loops are more than enough for WSN applications. It is not the focus of this thesis to explore this question, however it is the opinion of the author that the choice of whether to support exceptions and threads (and other features that may have controversial views) should be up to the application developer and not the specification.

2.5.2 Code Conventions

One of Java's benefits is ease of programming. Code conventions King et al. (1999) were encouraged in aim of facilitating easy to read code. Implications of some of the code conventions and other programming practices can lead to an increase in overhead. When using more powerful processors the overhead can be justified, however, in sensor networks, more computation means longer active times which results in higher energy usage. Popular code conventions which incurs extensive execution overheads will now be highlighted.

The code conventions issued by Sun King et al. (1999) state "Don't make any instance or class variable public without *good reason.*" Instead, it is recommended to encapsulate the logic in methods which alter class instance variables. Let's analyse the difference between accessing public instance variables directly and indirectly using method calls. Figures 2.4 and 2.5 displays the bytecode required to set an integer instance variable to the value of 1 followed by retrieving the value of the instance variable which is set to a local variable for indirect and direct instance variable accesses respectively. Comments are denoted by "//".

As can be seen in the examples indirect access of instance variables requires the same bytecode required to access instance variables directly, plus some extra overhead to call the class methods and pass or return values to or from it. In our example the following extra bytecode instructions were required to be executed: 2 `aload`, 2 `invokevirtual` and 1 `iload` instructions. This could amount to double the execution overhead to perform the same actions. Thus, a 'good reason' to directly access class or instance variables is that it is more efficient, and when working with a limited power budget efficiency is king.

Figure 2.4: Example of bytecode to indirectly access instance variables using getters and setters.

```
//call MyClass.setMyInt(1)
aload_1
iconst_1
invokevirtual #4 //(MyClass.setMyInt)

//bytecode executed in setMyInt(int i)
//MyClass.myInt = parameter
aload_0
iload_1
putfield #2 //(MyClass.myInt)

//call MyClass.getMyInt()
aload_1
invokevirtual #5 //(MyClass.getMyInt)

//bytecode executed in getMyInt()
aload_0
getfield #2 //(MyClass.myInt)

//bytecode executed in caller
//which assigns the return value to a local variable
istore_2
```

Figure 2.5: Example of bytecode to directly access instance variables.

```
//set field to 1
aload_3
iconst_1
putfield #8 //(MyClass.myInt)

//get field
aload_3
getfield #8 //(MyClass.myInt)
istore_2 //store it in a local variable
```

Other code conventions and programming practices which impose higher execution overheads may exist. It is important that application developers are provided with new code conventions which provide priority to efficiency.

2.6 JVMs for WSNs

The above questions which features of Java are relevant for WSNs. Now related work will be presented in respect to what has already been done in supporting JVMs for WSNs. The first proposed JVM for WSNs, Squawk, was proposed by Shaylor et al. (2003)

initially intended for smart cards having 32 bit processors and 160 KB of program space but later ported to the Sun SPOT (Simon et al., 2006) platform. The Sun SPOT is equipped with 512 KB of RAM and 4 MB of flash memory, making it substantially larger than the class of wireless sensor node devices targeted in this work. Several decision choices made in the Squawk system are however relevant to this work. More recent JVMs proposed for traditional WSN class devices (having tens of kilobytes of program space and ten kilobytes of RAM) include the TakaTuka VM (Aslam et al., 2008), the Darjeeling VM (Brouwers et al., 2008a) and the Mote Runner VM (Caracas et al., 2009). In this section we will discuss design choices and details which are relevant to the work being presented in this thesis.

2.6.1 Application Portability

One of the design goals behind Java is application platform independence. By employing a bytecode encoding the virtual machine can then execute the logic on the underlying hardware (either using an interpreter or by compilation techniques). The JVMs proposed for WSNs all make use of a bytecode encoding which allows for application portability, meaning that the same bytecode can be interpreted on sensor nodes with differing hardware architectures.

2.6.2 Pre-processing Bytecode

The Sun SPOT sensor node targeted by the Squawk JVM has substantially larger resources than the nodes targeted by this work and that of the other proposed JVMs for WSNs. However still much smaller than the resources available in traditional computing platforms. Shaylor et al. (2003) initially proposed an offline class file pre-processing stage which above all transforms bytecode produced by the standard Java compiler into more compact bytecode to better suit the resource constraints. Later Simon et al. (2006) proposed the term split VM architecture, whereby the offline pre-processing stage involves loading, verification and transformation of Java bytecode on a host machine. The device is then only required to store the pre-processed bytecode. Thereby the device and host machine are seen as a single large virtual machine, providing different parts of the virtual machine bytecode compilation and loading process. Due to the increased constraints on the platforms targeted by the other JVMs, offline pre-processing is also included in their designs which can also be viewed as a split VM architecture.

2.6.3 Compact Bytecode Instruction Set

Due to the limited program space available it is beneficial to minimise the overhead required to store application logic. Therefore, the Java bytecode produced from the

Java compiler is pre-processed and translated to a more compact bytecode specification that better suits the target platform. Where possible, Shaylor et al. (2003) replace Java bytecode instructions that require two operands to require only one operand. An example of this includes branch instructions which provide a two byte destination address. If the destination address fits in a byte, then essentially the second byte is not used. The `iload` bytecode instruction is used to load an integer local variable, whereby a single byte identifying the local variable index to load follows the `iload` instruction. To reduce size, the Java bytecode specification (Lindholm and Yellin, 1999) also includes bytecode instructions which combine the `iload` instruction with a variable index number, from `iload_0` to `iload_3`. In aim of further decreasing code size, Simon et al. (2006) propose increasing more combined single byte bytecode instructions of this type for load, store and const related instructions. Similar to a single byte operand address for branches, the same is proposed for field and method addresses. To facilitate more efficient Garbage Collection (GC), the bytecode is altered so that local variables are re-allocated such that value slots are separated from reference slots, and therefore one pointer map is required per method.

2.6.4 Bytecode Optimisation

Simon et al. (2006) propose inlining getter and setter methods, as well as small static or final methods. The authors also propose constant folding and constant propagation. Constant folding involves resolving operations at compile-time if the sufficient operands are available at run-time. Constant propagation involves replacing variable value usage with that of a constant value, if the value of the variable can be determined at compile-time. By inlining code more constant folding and propagation optimisations should be possible, especially due to Java's recommended getter and setter methods for accessing variables. Consider the following code as an example of code before constant folding and propagation:

```
int minute = 60;
int hour = 60 * minute;
int day = 24 * hour;
```

and the resultant code after constant folding and propagation:

```
int minute = 60;
int hour = 3600;
int day = 86400;
```

Bytecode size reduction techniques are usually split up into two categories: compression and compaction (Beszédes et al., 2003). Compression techniques that unpack the underlying bytecode on loading could be useful, however if the decompression stage was

to be implemented during execution this would greatly hinder the execution overhead already inherent in interpreters. Aslam (2011) proposes a compaction technique that utilises unused bytecode instruction values to represent common bytecode instructions used. They consider operand reduction, operand removal and compacting multiple instructions together. Operand reduction is the process of reducing the operand size. Java instructions that point to branch offsets contain a 2 or 4 byte operand whilst constant pool indexes contain a 2 byte index. Aslam (2011) propose to reduce the operand size for operands having values less than 256. More so, to optimise the constant pool access they also order the constant pool entries according to the most commonly used. Operand removal involves the complete removal of an operand. This can be done by merging a bytecode instruction and an operand to a single bytecode instruction value. Furthermore, they propose to identify popular sequences of bytecode instructions that can be compacted to a single bytecode instruction (with a number of operands). During compilation they analyse the bytecode for the different compaction optimisations and replace the most common optimisation with the respective compaction. Code to handle the decompaction is appended to the interpreter loop and therefore the compaction scheme is dynamically changed according to the application being compiled.

2.6.5 Symbolic Name Resolution

Java class files contain symbolic class, method and field name information. These names can allow dynamic run-time loading and inspection of class files and objects (most often through Java reflection libraries). When resolving names a lookup is required to find the actual address of the associated element. Many Java interpreters perform this lookup whilst executing code. In order to optimise this process, the first time a lookup is performed, the bytecode is patched to include a direct reference. As pointed out by Shaylor et al. (2003) patching bytecode on the devices in question typically requires rewriting flash memory (which is time consuming and incurs substantial energy). In respect to the target platform however, this textual representation may be more of an overhead than a benefit. By resolving the symbolic name information, execution is increased by replacing the symbolic name with the actual address. This approach is used by the Squawk (Shaylor et al., 2003) and Darjeeling (Brouwers et al., 2009). The Squawk JVM originally allowed this translation to either be done offline during the pre-processing stage, or on node during the loading phase. However this was changed to only implement name resolution during the pre-processing stage. The decreased bytecode size via symbolic name resolution, comes at the cost of loss of reflection, however reflection is a rarely required for WSN applications.

2.6.6 Language Independence

The approach proposed by Caracas et al. (2009) provides a Java or C# based programming environment. The bytecode pre-processing stage they present can take as input either Java bytecode or Common Intermediate Language (CIL) instructions (ECMA International, 2006) generated from languages implementing the .NET specification (Miller and Ragsdale, 2003) such as C# (Hejlsberg et al., 2003). This could be possible in the other proposed solutions as well provided that a translator is created to translate from the input instruction set to the platform's instruction set and given that both languages support stack based architectures this should not prove to be a problem. However, in light of programming language research this would prove to be an asset in that new language translators could easily be implemented and tested for resource constrained devices.

2.6.7 Garbage Collection

Java abstracts the memory allocation and de-allocation task from the developer. Thereby garbage, i.e. memory reserved by the application but no longer used, gradually increases. Therefore, Java requires a garbage collector to release such reserved memory. The main control loop for the Squawk JVM is:

```
for(;;) {
    interpret(result);
    result = gc();
}
```

The `interpret` function will exit only when memory allocation fails. The garbage collector will then be executed and remove any memory reserved that is no longer being used. Following this if the available memory is less than that which is required by the memory allocation instruction which caused the allocation fail, then the garbage collector will return false and then the interpreter will then issue an error. If memory is available, then execution will continue as normal. Garbage collection can only occur on a method entry since a stack frame is allocated at the beginning of a method, when creating a new object (whereby the memory allocation is performed in the object's constructor) and an explicit call to the `System.gc` method. Garbage collection typically involves traversing memory and freeing any memory which was previously used but is no longer reachable from the program. The process of executing garbage collection during program execution is referred to as *online* garbage collection. Different online garbage collection techniques used within the WSN targeted JVMs will now be discussed followed by a more recent offline garbage collection scheme.

2.6.7.1 Mark and Sweep GC

McCarthy (1960) proposed the first automatic storage reclamation technique, the Mark and Sweep algorithm. The algorithm's principle is to allow memory to be allocated from the heap without requiring explicit de-allocation or techniques to automatically de-allocate memory when an object is no longer used. When an allocation is required and the heap returns that not enough memory is available, then a traversal on the heap will be made to determine which objects are reachable, and which ones can be de-allocated. After marking reachable objects as active, all the unmarked (unreachable) objects will be swept, i.e. de-allocated. This technique is used by the Darjeeling VM.

2.6.7.2 Mark and Compact GC

During a program's life cycle, memory will be allocated, used and then de-allocated. When memory is de-allocated it leaves a gap of free memory. The gaps will continue to get smaller and smaller over a program's lifetime. This could eventually result in cases where enough memory is available however is fragmented and thus a contiguous amount of required memory cannot be allocated. Mark and Compacting Garbage Collection (Jones and Lins, 1996), used by the TakaTuka VM, can alleviate this problem by separating memory into two sections, one containing live data and the other section containing memory that is free to use. Mark and Compacting algorithms consist of three main phases. The first phase consists of traversing reachable objects and marking each object as reachable, the second phase then consists of relocating objects so that they all fit two one side of the heap and the final stage involves resolving and updating any pointers that reference to objects that have been relocated.

2.6.7.3 Generational GC

When performing garbage collection the heap must be traversed to determine which objects are unreachable and therefore garbage. Objects living at the end of the traversal tree will require more time to traverse to in comparison to objects living at the beginning of the traversal tree. It is widely believed that "most objects die young" (Ungar, 1984). Generational garbage collection is designed around this, in that de-allocation can be made more efficient if it can find the objects that are more likely to die younger quicker. Generational GC splits the heap into two or more parts, each associated with a generation, whereby a generation is defined by an object's age. Garbage collection can then be performed first on only the younger generation, and only if more memory is still required after GC, then further GC can be performed on the next older generation. This technique is employed by the Squawk VM along with a Mark and Compact algorithm.

2.6.7.4 Offline GC

Aslam (2011) identifies two main drawbacks with online garbage collection for resource constrained devices. The first being that if an object is reachable from the program but is never actually used, the garbage collector will not be able to de-allocate the memory associated with this object. The second drawback is that garbage collection will frequently be invoked which results in a traversal on the object graph and ultimately in an increased execution overhead and reduced lifetime. Offline analysis techniques proposed for traditional computing platforms (Choi et al., 1999; Blanchet, 2003; Guyer et al., 2006; Albert et al., 2007; Cherem and Rugina, 2007) attempt to reduce the execution overhead required to perform garbage collection in a system, but do not provide any mechanisms to de-allocate objects that will not be used that are still reachable. Aslam (2011) propose a method to de-allocate objects that are reachable but that will not be used by performing analysis at compile time and therefore name this scheme Offline GC. Analysis is performed during compilation to determine when objects can be explicitly de-allocated (since they will no longer be used). The approach introduces new bytecode instructions to instruct the VM to de-allocate specific objects. Once the objects to be de-allocated are identified, the new bytecode instructions can be inserted in the pre-processed bytecode to explicitly instruct the VM to de-allocate the objects. A normal online garbage collection scheme must also be used in conjunction with offline garbage collection. The benefits of doing so are that more memory is made available to the application (since other techniques do not release objects that are still reachable even if they will never be used) and as a result of this garbage collection will be required to be executed less.

2.6.8 Threading

Interpreters keep track of the current executing instruction by storing a bytecode program counter which points to the next address to execute. Squawk originally supported threading by assigning a counter to each thread. The counter is decremented "every time the interpreter executes a branch" (Shaylor et al., 2003). When the counter reaches zero a thread switch is performed, which involves saving the state of the current thread (such as the bytecode program counter). Later, Squawk was altered to support Green Threads. Green Threads implemented threading without relying on native OS threading libraries or features, and therefore run in user space and not in kernel space. Green Threads do not support pre-emption, but instead perform thread switching either when explicitly instructed by code (using `Thread.yield()`, `Object.wait()` or other methods that imply a thread switch) or when a blocking operation is executed.

2.6.9 Debugging

Squawk supports debugging of applications on Sun SPOTs by using the Java Debug Wire Protocol (Sun Microsystems). JDWP is a protocol which is used to communicate with a JVM to inspect and debug applications that are running on top of the JVM. This involves reading class file information (including fully qualified names, source line tables and other class meta-information). Since this information is stripped out by the pre-processor approach proposed, JDWP cannot be implemented directly to the target devices. Instead, by using a similar approach to the split VM architecture, a split debug architecture is also used whereby a debug proxy runs on a host machine. Class files are loaded into the debug proxy, and thereafter the proxy retrieves state information from the device and can reconstruct the whole debug information state by merging the current state information and the class file information. Simon et al. (2006) states that the overhead introduced when enabling debugging is 10%. Aslam (2011) also propose a similar approach to that of Simon et al. (2006). Aslam (2011) implement a solution that requires less RAM due to a native C implementation when compared to the Java implementation proposed by Simon et al. (2006) and also a light-weight wire protocol to decrease RAM requirements.

2.7 Interpretation is slow

Java initially provided JVMs based purely on interpretation and the performance was notably slow, as worded by Cramer et al. (1997) "Interpreting bytecodes is slow.", and by Tyma (1998) "Java isn't just slow, it's really slow, surprisingly slow." and by Aycock (2003) "less-than-stellar performance". Initiatives towards increasing the performance of Java was therefore conducted. The following sections provide an overview of different techniques proposed to increase the execution performance of Java.

2.8 Java Processors

Noting that interpreters were inefficient and that effective dynamic compilers for Java required hundreds of kilobytes of program space and megabytes of RAM, McGhan and O'Connor (1998) proposed a processor architecture, PicoJava, that can natively understand Java bytecode and thereby achieve highly efficient execution of bytecode without incurring any overheads due to a run-time environment. Target devices for this work included "set-top boxes, smart phones, mobile phones, PDAs and other handheld devices, automotive systems, smart controllers, smart cards, and so on." Cormie (2000) propose Jazelle, an extension to ARM CPU cores to support an extra mode that operates on bytecode. These CPUs have substantially more resources than the microcontrollers used

in WSNs. Ito et al. (2001) claims that PicoJava does not address resource-constrained applications, perhaps due to the cost involved in a larger processor, and therefore identifies the need for Java microcontrollers. Ito et al. (2001) demonstrate the FemtoJava architecture which provides a Java microcontroller synthesized within an FPGA. A WSN node based on the FemtoJava processor, FemtoNode, was proposed by Allgayer et al. (2009). Hardware implemented VMs can be provide a more efficient Java execution platform which ultimately results in less power consumption, however there may be barriers to its adoption since the target audience is limited, as can be seen from the popularity of Java specific CPU based processors. Lower production volumes would result in higher costs and therefore would not be relevant for such applications. However, this is definitely an area of interest where further research and marketing initiatives should be taken.

2.9 Compilation, Interpretation and Semi-Compilation

Programming languages (higher than machine code) are translated and executed using two different schemes being interpretation and compilation. Interpretation is often attributed by its slow execution speeds, whilst compiled code executes natively on the underlying hardware and therefore executes faster. The JVMs proposed for WSNs and early JVMs for traditional computing execute instructions by means of interpretation. Compilation techniques aimed at increasing execution of Java bytecode will follow, and in order to provide the reader with a better understanding of compilation and interpretation a brief introduction will now be provided.

Computing devices can understand machine code (also referred to as native code). Machine code programs consist of instructions that are encoded in a binary format which is very hard for humans to understand. To facilitate humans to be able to program computing devices easier programming languages with human readable instructions were created. However, computing devices cannot understand anything but machine code. Therefore, the programming language code needs to be translated to machine readable native code. A compiler is a computer program which translates code from a source language (or a programming language) to a target language, usually with the intent of executing the program logic on a computer (or other device). This translation process is called compilation.

Many different dialects or types of machine code languages exist. A computing device will usually only understand one type of machine code. When compiling from a source programming language for a device, the target machine code type must be specified. This means that the resultant target machine code can only execute on devices that understand that specific instruction set of machine code. Therefore, the same programming language would have to be recompiled for the different target platforms it is intended to

support. A solution to recompiling code for different target platforms lies in interpretation. An interpreter is a computer program that can understand and translate a source language and execute it on the target platform. By installing an interpreter on each different computing device, recompilation of the source code for each target platform would not be required. Interpreters are also useful for environments where compilers are not available and code is required to be changed (perhaps on-the-fly). Interpreters incur extra processing overheads since the interpreter has to translate the input language, whilst code compiled to machine code is natively understood by the underlying hardware.

The difference between compilation and interpretation is clear, compilation is the process which translates code from the source language to the target format prior to execution, whilst interpretation translates and executes the code on-the-fly. Using only compilation or interpretation is sometimes not ideal. High level programming language code is highly abstracted, therefore interpretation would involve extensive processing overheads. Thus, a hybrid solution is often used called semi-compilation, whereby code is compiled from the high level source language to a lower level intermediate language but is not targeted to a specific machine code instruction set. The advantages of this is that the code can be easier translated in comparison to the high level source language. Modern high level languages such as Java and C# use semi-compilation whereby the input language is compiled to Java bytecode or .NET CLI code respectively. The code is then interpreted (or compiled) on the specific platform as required. This chapter will now conclude with an overview on different compilation techniques.

2.10 Way Ahead of Time Compilation

To increase Java performance Dean et al. (1996) proposed compiling Java (and other high level languages) to C code which is in turn compiled to native code. A similar approach was taken by Proebsting et al. (1997) in which they term the compilation process as Way Ahead of Time (WAT) compilation. This process is often described Ahead-Of-Time (AOT) compilation as well. However, for the purpose of clear separation between the time in which code is compiled whilst developing and loading time, throughout this thesis the term Way Ahead of Time will be used to describe compilation during development, and Ahead-Of-Time to describe the compilation process that occurs after loading code (and before execution time). Other approaches that implement WAT compilation by first converting to C include Gilles Muller (1996); Muller et al. (1997); Thomm et al. (2010). Courbot et al. (2010) propose a 'romization' method for embedded systems whereby Java code can execute offline on an external system and thereafter translate the code and system state into a single C file which is then compiled to a native code image and 'burned' onto the end device. The GNU Compiler for Java (GCJ) (Bothner, 2003) provides WAT compilation, however instead of translating the produced bytecode

to C, it treats Java like any other language using GCC and compiles it directly to native code. The approaches demonstrate a reasonable improvement in speed, however at the sacrifice of platform independence. In respect to WSNs, although this may not be an issue when initially deploying code, platform independence would be sacrificed and more so update sizes are likely to be larger due to the nature of native code.

2.11 Ahead-Of-Time Compilation

WAT compilation although potentially able to produce highly efficient code (due to offline analysis and compilation) strips the VM of platform independence. Ahead-Of-Time (AOT) compilation on the other hand, performs compilation when code is loaded into the system. Thereby it can generate efficient code prior to execution whilst at the same time not sacrificing platform independence. Hsieh et al. (1996) demonstrate a simple AOT compilation technique which translates bytecode to native code by mimicking the Java operand stack. Alpern et al. (1999) present Jalapeo, a JVM written in Java that uses an AOT compilation paradigm. Alpern et al. (1999) describe three compilers, a baseline compiler which mimics the Java operand stack (similar to Hsieh et al. (1996)), an optimizing compiler (Burke et al., 1999) to speed up computationally intensive methods and a middle ground which performs low level optimisations (primarily register allocation). Execution efficiency is highly dependent on the optimisations performed, and therefore although WAT compilers may be able to include more optimisation if resources are limited on the target device, AOT compilation still provides sufficient execution efficiency without sacrificing platform independence. A simple baseline compiler which mimics the Java stack should require footprint comparable with that of an interpreter since roughly the same translation logic is required. This work serves as initial motivation towards implementing run-time compilation of bytecode in severely resource constrained devices. Since an AOT compiler translates bytecode to native code during loading, more program space will be required when compared with a bytecode encoding, and in severely resource constrained devices this may be a problem (depending on the application).

2.12 Just-In-Time Compilation

Just-In-Time (JIT) compilation translates code from bytecode to native code during program execution, and therefore it maintains a low program space requirement along with a native execution paradigm without sacrificing platform independence. Although JIT compilation is often a technology that is associated with Java, as pointed out by Aycock (2003), it has been around since the early 1960's with McCarthy's (1960) LISP paper. Throughout the years JIT compilation techniques have resurfaced for systems such as Smalltalk (Deutsch and Schiffman, 1984) and Prolog Haygood (1994).

Cramer et al. (1997) describes early work on JIT compilation and motivation for further work to enhance efficiency of JIT compiled code. Extensive research was proposed for improving JIT compilation by including common subexpression elimination, efficient register allocation, array bounds check elimination, basic block analysis, static method inlining, stack analysis, dynamic and hybrid compilation/interpretation techniques and more (Plezbert and Cytron, 1997; Adl-Tabatabai et al., 1998; Krall, 1998; Ishizaki et al., 1999; Yang et al., 1999; Suganuma et al., 2000, 2001). Stripped down JVMs and JIT compilation was proposed for mobile device class embedded devices having a program space footprint size of larger than 140 kilobytes (much larger than the device class proposed in this thesis) (Shaylor, 2002; Delsart et al., 2002; Gal et al., 2006; Debbabi et al., 2004, 2006). It is most likely due to the large footprints required (by even JIT compilers targeted for mobile class devices) that it is believed that JIT compilation is something beyond the power of WSN class devices.

2.13 Basic Block JIT

A basic block is a unit of code that has one entry point and one exit point "without having the possibility of branching except at the end" (Aho et al., 1986). JIT compilers traditionally translate code at the granularity of whole methods. When code is being executed it is possible that certain code paths are not executed due to conditional and branching statements. Rogers (2002) propose Basic Block JIT compilation to increase system performance by only compiling code that will be executed. To achieve this, a JIT compiler must first perform basic block analysis to determine basic block starting and ending points, and thereafter compile basic blocks as they are encountered. Although originally proposed for an increase in efficiency, basic block compilation can serve as a means of minimising memory overhead when performing compilation, and therefore this technique is further investigated in this thesis.

Chapter 3

The Case for Run-time Compilation of Bytecode in Wireless Sensor Networks

As demonstrated by Brouwers et al. (2009) interpreting bytecode incurs a high execution overhead compared with native code. Although it can be argued that sensor nodes sleep most of the time and even that execution speed of non-time critical programs is less important (Brouwers et al., 2008a), the fact that the interpretation overheads result in higher power consumption for executing code compared with a native code application cannot be dismissed. In this chapter we investigate the consequences of interpretation on a sensor node's lifetime and provide a case and motivation for research towards run-time compilation techniques by modelling the overheads inherent in interpretation and also by performing an experimental analysis of the effects of interpretation on timing and power consumption.

3.1 Modelling Interpretation Overheads

The cost of executing bytecode can broken down as follows. The execution cost of interpreting and executing a Java bytecode instruction, $E(i)$, can be expressed as:

$$E(i) = E_{int}(i) + E_{stack}(i) + E_{exeop}(i) \tag{3.1}$$

where i is the bytecode instruction to execute, $E_{int}(i)$ is the overhead required to interpret the bytecode instruction, $E_{stack}(i)$ is the stack operation overhead associated with the respective bytecode instruction and $E_{exeop}(i)$ is the overhead required to execute the arithmetic and logical operations associated with the bytecode instruction.

In aim of minimising the cost of bytecode execution, it is proposed to remove the interpretation cost, $E_{int}(i)$, from the 'fetch, decode, execute' cycle. A typical interpreter implementation will consist of a switch statement with a case code block for each possible bytecode instruction. The switch statement for the Darjeeling virtual machine was analysed (or any virtual machine really) and the resultant native code generated for the MSP430F1611 microcontroller (which is commonly used in WSN nodes). The cost associated for a switch statement is at least 5 native code instructions (i.e. $E_{int}(i) \approx 5$), depending on the build optimisation settings, for each bytecode instruction. Obviously an interpreter would involve a significant amount of other interpretation related overheads (substantially larger than 5 native code instructions). In fact, Mitchell (1970) state in their early work on translation from Smalltalk to native code that "the benefit of even this simple kind of translation will be great."

The Java bytecode instruction set is inherently stack based and thus it has been decided to sustain the stack operation overheads, $E_{stack}(i)$, associated with bytecode execution in return for a smaller sized run-time compiler.

Thus, the energy consumption of the proposed method should amount to $E_{stack}(i) + E_{exeop}(i)$, therefore removing a minimum of 5 native code instructions (although the reduction will be much larger) from the execution overhead for each bytecode instruction.

Furthermore, although it is widely believed that run-time compilation is non-trivial, infeasible, or impossible on such small devices (Palmer, 2004; Koshy and Pandey, 2005; Pandey and Koshy, 2006; Koshy et al., 2008; Aslam, 2011), the thesis presented is that a simple run-time compiler can be achieved with roughly the same amount of code that is required for an interpreter since the same translation is required to be done anyhow.

The general consensus in the wireless sensor networks community is that the execution overheads introduced from an interpreter are of minor importance (Brouwers et al., 2009). This may seem especially true when the dissemination costs of firmware is factored in as presented by Steinfeld and Carro (2009). This is due to the decrease in overall overhead since dissemination costs of bytecode is smaller than that of native code. However, if it was possible to implement a platform which would allow a bytecode dissemination encoding and a native code execution platform, dissemination costs could be reduced to that of an interpreter equipped sensor network whilst at the same time minimizing the execution overheads inherent in an interpreter. Now, the question is whether the interpretation overhead is significant to a sensor node's lifetime.

To put this into perspective, a model of the expected lifetime of a sensor node based upon the duty cycle and the current consumption of the microcontroller's execution and sleep states will now be provided. The expected lifetime of a sensor node, L, is defined as:

$$L = \frac{B}{\sum_s I_s T_s} \tag{3.2}$$

where B is the total battery capacity, s represents a microcontroller power state, I_s is the microcontroller's current consumption at state s, and T_s is the ratio of time at which the microcontroller is in state s. Obviously, a wireless sensor node would also incur sensing and communication overhead. However, let us for now ignore the sensing and communication overheads in aim of emphasizing the significance of interpretation costs. Let us also assume that the microcontroller exhibits two power states, an active state and a sleep state, since interpretation will only affect the duration of the active state. Thus, (3.2) can be simplified to two states as follows:

$$L = \frac{B}{I_{active} \cdot T_{active} + I_{sleep} \cdot (1 - T_{active})} \tag{3.3}$$

where I_{active} and T_{active} are the current consumption and ratio of time spent in the active state for native code execution and I_{sleep} is the sleep state current consumption.

Execution speeds for an interpreter demonstrated by Brouwers et al. (2009) ranged from 30.4 to 113.2 times the speed of the native code implementation. Let us assume that the interpreter can achieve an optimal execution speed of 30 times that of native code. Thus, T_{int}, the ratio of time in which the microcontroller is in an active state for an interpreter can be defined as:

$$T_{int} = T_{active} \cdot 30 \tag{3.4}$$

and L_{int}, the lifetime of a node equipped with an interpreter can then be defined as:

$$L_{int} = \frac{B}{I_{active} \cdot T_{int} + I_{sleep} \cdot (1 - T_{int})} \tag{3.5}$$

Using (3.3) and (3.5) the lifetime of a node for a varying duty cycle is plotted in Figure 3.1 for native code and interpreted implementations. The degradation factor (i.e. the native version's lifetime divided by the interpreted version's lifetime) is also presented which highlights the performance loss when using interpretation. Let us assume a battery capacity, B, of 3,000 mAh, and use current consumption levels for the Telos B sensor node, i.e. an active state, I_{active}, of 1.8 mA and a sleep state, I_{sleep}, of 5.1μA. As can be seen from Figure 3.1, as the active percentage of the duty cycle increases, the lifetime of the native code implementation compared with the interpreted version approaches the interpretation overhead (in this case 30). Even with a low active duty cycle of 0.1% (for example, 3.6 seconds per hour), a native implementation will last 8.5 times more than an interpreted implementation. The model only considers the microcontroller's current consumption for active and sleep states, and therefore results in an extremely long lifetime which is not realistic. This is due to the model not factoring in hardware related issues such as battery dissipation; and other sources of energy consumption typical to sensor nodes including sensors and transceivers which essentially results in

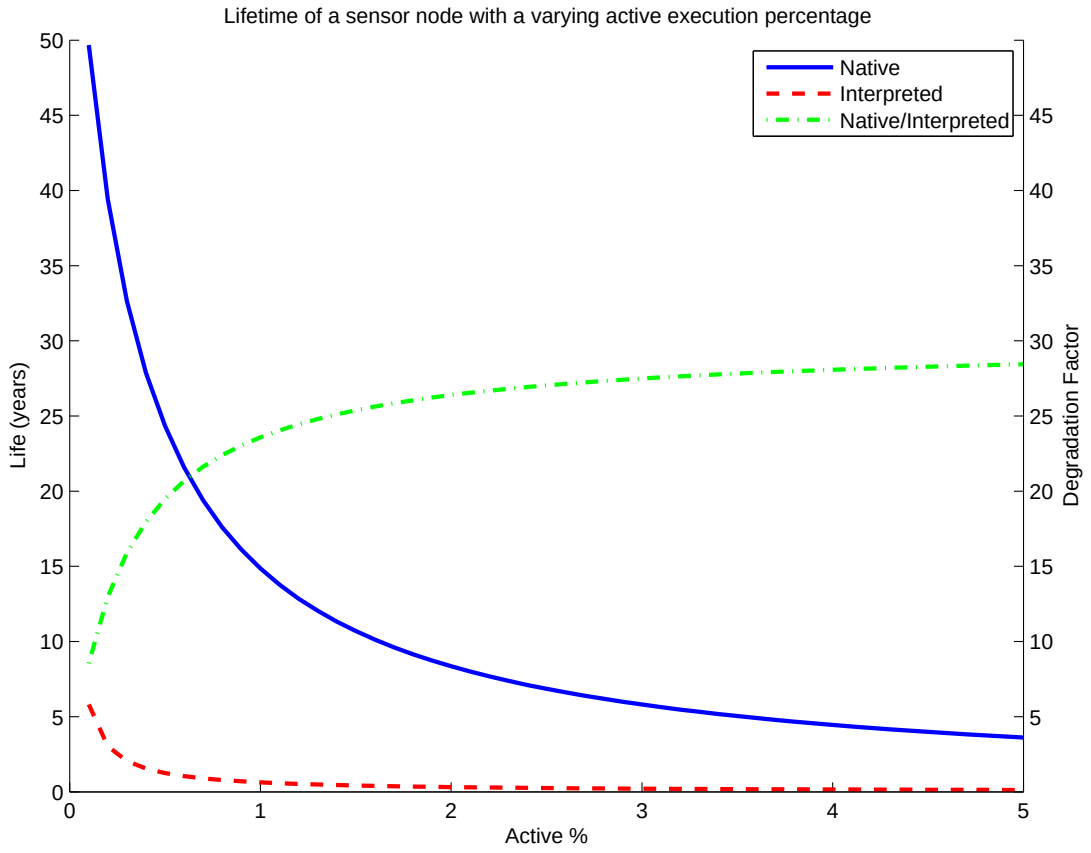

Figure 3.1: Lifetime of a sensor node with a varying active execution ratio for native code and an implementation with 30 times the overhead. The degradation factor (native lifetime divided by the interpreted lifetime) also highlights the performance loss when using interpretation. As the active percent increases the degradation factor tends toward the overhead, i.e. in this case 30.

a sensor node that cannot sense or transmit. The purpose of the model, however, is simply to demonstrate the impact of execution overheads in isolation.

Sensor networks, however, do not just consist of active and sleep components in a duty cycle, they also include a sensing and communication component. Let's now consider adding sensor and communication costs to the model. The lifetime of a sensor node is obviously dependent on the application logic which controls the amount and timing of computation, sensor acquisition and communication tasks. Obviously, each different possible configuration cannot be modelled since there exist infinitely many. However, as can be seen from Figure 3.1, as the active percentage of the duty cycle increases, the factor at which the native code implementation lasts longer when compared to the interpreted implementation increases (and approaches the overhead). Thus, modelling on those applications which incur a minimal active duty cycle percentage, such as a sample and send application will be provided to highlight the performance of best case scenarios of interpretation (since worst case applications will only prove the case even more).

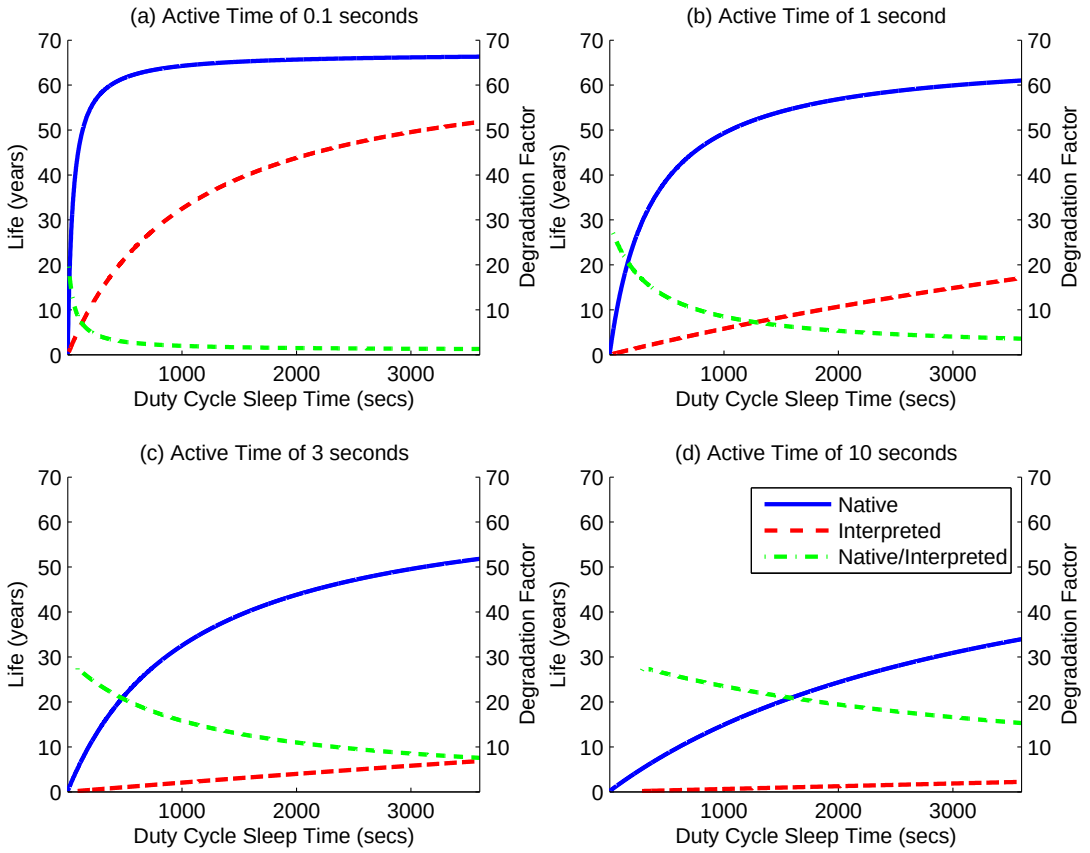

Figure 3.2: Lifetime of a sensor node for an application with a varying duty cycle including a single sensor acquisition and communication transmission for an active time of (a) 0.1, (b) 0.5, (c) 1, (d) 3, (e) 5 and (f) 10 seconds.

Let's define E_{cycle} as the energy consumed in a single duty cycle for a sample and send application as follows:

$$E_{cycle} = (Cycle_{sleep} \cdot E_{sleep} + Cycle_{active} \cdot E_{active} + E_{sensor} + E_{radio}) \tag{3.6}$$

where $Cycle_{sleep}$ and $Cycle_{active}$ are the lengths of the sleep and active components of the duty cycle respectively; and E_{sleep} and E_{active} are the energy consumption values for the sleep and active states respectively. Using 3.6, the expected lifetime for a given duty cycle can now be computed by dividing the energy available, $E_{battery}$, by the energy consumed for a single duty cycle, E_{cycle}, multiplied by the duration of a single duty cycle as follows:

$$L = \frac{E_{battery}}{E_{cycle} \cdot (Cycle_{sleep} + Cycle_{active})} \tag{3.7}$$

By using 3.7 the expected lifetime of a sensor node can now be estimated for a given duty cycle with a fixed sensor and communication cost. Let's assume the same values as

above for current consumption and a 3.3V battery source, therefore the energy required during sleep, E_{sleep}, is 16.83 μJ and 5.94 mJ whilst active, E_{active}. Also, let's assume that E_{sensor} is 80 μJ and E_{radio} is 81.6 μJ (typical values for a 12-bit reading using an SHT11 (Sensirion, 2010) and a 100 byte length message being sent over the air using a CC2420 radio (Texas Instruments, 2010) at -25 dBm). $E_{battery}$ can be calculated to 35640 J from a 3300 mAh battery and a 3.3V battery. Figures 3.2 (a), (b), (c), and (d) depict the expected lifetime in years against the sleep duration of a duty cycle in seconds for an active duty cycle duration of 0.1, 1, 3 and 10 seconds respectively using equation 3.7.

As can be seen from the graphs in Figure 3.2, the degradation factor which is imposed by using interpretation gradually decreases as the amount of sleep time increases. As can be noted from 3.2 (d), the interpreted lifetime prior to a 280 second sleep time is not depicted (and therefore the degradation factor is also not depicted). This is due to the fact that the interpreted version prior to the cut-off point requires more time to execute the required computation than the time available in the whole duty cycle and therefore the node cannot enter the sleep component of the duty cycle. The reason why the sleep component of the duty cycle is less for the interpreted version is that it is assumed that the quality of service and requirements of the sensor network does not change according to the execution platform, however is inherent in the specific sensor node application. The model once again reports extremely high lifetimes which is due to the simplistic model used, and more so due to the extremely low active computation times, extremely long sleep times, and other factors that are not accounted for such as battery dissipation. However, the purpose of the model is to show that even for very low active cycles and very long sleep cycles, interpretation overhead will greatly hinder efficiency. The higher the active component of the duty cycle, the greater the degradation factor of interpretation will be.

3.2 An Experimental Analysis of the Effects of Interpretation

From the model above it is shown that as the amount of computation increases, the interpretation degradation factor increases. It is worth noting that in an interpreted system not all execution will take place by means of interpretation but also by means of native execution for natively implemented drivers and libraries. The more execution that takes place natively, the less the interpretation costs. This is obviously highly dependent on the actual user application code being executed and also on how the specific run-time system was implemented. In order to evaluate whether or not the execution overheads will be high enough to justify the research proposed in this thesis, an initial evaluation of an existing virtual machine, namely the TakaTuka virtual machine (Aslam et al., 2008), was undertaken. The overheads inherent in execution were high compared to native code

Table 3.1: Radio On/Off Evaluation

	Turn On Time (ms)	Turn Off Time (ms)	On-Off Consumption (mJ)
TakaTuka VM	5.2	2.3	0.159
Native Code	1.6	0.011	0.02

Table 3.2: Toggle LED Evaluation

	Toggle Time (μs)	On-Off Consumption (μJ)
TakaTuka VM	1261	16.5
Native Code	5.2	0.11

as expected and reported by the authors. Execution overhead experiments are further described in Chapter 5. Initial experimentation of the interpreter not only revealed the high execution costs, but more interestingly the indirect additional energy expenditure due to longer execution times.

WSN applications aim to turn off peripheral hardware as much as possible to lower energy consumption. Therefore, a first experiment was performed to establish the time taken to turn on and off a radio transceiver and also the energy consumed in doing so. A Telos B (Polastre et al., 2005) sensor node was connected to an oscilloscope to monitor current consumption. An application to continuously turn on and off the radio was loaded for both the TakaTuka VM and a natively coded application. The results are presented in Table 3.1. The time taken to turn the radio on and off is presented along with the consumption for a single on-off cycle. The time taken to turn the radio on is 3.25 times longer for the TakaTuka VM than compared to a natively coded implementation. Turning on the radio involves enabling the radio's power line followed by configuring the radio's internal registers. The TakaTuka VM takes 209 times longer to turn the radio off, which involves switching off the power line, however additional work could be performed here such as configuring the internal registers into a sleep mode. Actually, this comparison is not exactly a fair comparison since it is highly dependent upon the radio driver implementation. This is most likely why large differences are seen between the comparison of turning the radio on and turning the radio off. Nonetheless, it does show higher energy expenditure even if it is due to poor driver implementation. The energy consumption required to turn the radio on and off once using the virtual machine takes 7.95 times than that of native code. Again, although this is highly dependent on driver implementation, for this particular case the virtual machine implementation will result in much higher energy costs not only for the computation part of a duty cycle, but also for the usage of the radio.

To evaluate the indirect costs fairly, instead of using a radio transceiver, it was decided to analyse the difference required to turn on and off a LED. A LED is turned on or off by flipping a bit of an internal register that controls the output pins, which can be implemented with only several instructions. It is therefore hard to blame driver implementation for any extra energy consumed when switching on and off a LED. The same experimental setup for the radio test was used, except instead of turning the radio

on and off, a LED was toggled. The results are presented in Table 3.2. The time taken between LED toggles for the TakaTuka VM is 230 times longer than that of a native implementation, and therefore consumes more energy since the actual device is on longer. The energy consumption required to turn a LED on and off once is 150 times that of native code. Again, this will greatly impact the lifetime of a sensor node since not only will the node have to deal with the burden of longer computation times and the resultant higher computational energy costs, however it will also incur higher energy costs for peripheral hardware usage. The reason why there is a such a higher energy consumption for using hardware lies in the fact that it takes the interpreted code longer to turn on and off devices. Once a device is powered on, the longer it takes to perform any required processes and turn off the device, the higher the consumption will be.

In this chapter an analysis on both the direct and indirect costs of interpretation has been provided. It is evident that the lifetime is substantially decreased for an interpreted application when compared to a native code implementation. A high level object oriented programming language such as Java, however provides many development advantages. Thus effort should be put into achieving a Java execution run-time which does not interpret code, but executes code natively. Therefore this investigation provides grounds that run-time compilation of bytecode on sensor nodes should be further explored to evaluate primarily whether it is possible and feasible, and secondly to examine what overheads are associated with a run-time compiled application when compared to a native code application, and furthermore the effects of any incurred overheads on the lifetime of a sensor node.

Chapter 4

Enabling AOT Compilation for Resource Constrained Devices

In the previous chapter the costs of interpretation and native execution are modelled. A native code platform can provide a substantially more efficient execution environment. Virtual machines however can enable higher level abstractions to support ease of development. In aim of supporting both a VM abstraction and a native code execution platform, this thesis proposes run-time compilation techniques which are deemed as something beyond the power of WSN class devices. In this chapter techniques to enable an Ahead-Of-Time (AOT) compilation are presented.

4.1 Design

A Java programming paradigm can provide many advantages for programming WSNs, however recent work on compatible virtual machines has revealed the high execution costs incurred when using an interpreter (Brouwers et al., 2008a). Therefore, an AOT compiler has been designed which compiles bytecode into the underlying platform's native code. The design directions and decisions that have been taken will now be discussed.

4.1.1 Requirements

Similar to Darjeeling, Mote Runner and TakaTuka the primary goal of this work is to achieve a Java bytecode execution platform for WSNs. However, the contribution presented here is focused on techniques to enable AOT compilation which aims to minimize the overhead required to execute bytecode on sensor nodes. The main requirements for the proposed approach are outlined as follows:

4.1.1.1 Ease of Use

The virtual machine should be augmented with an environment that allows programmers to develop applications for sensor nodes with minimal effort. New languages or paradigms should only be imposed if it is absolutely necessary. Thus, it has been decided that like the other approaches, to support Java bytecode in which a high number of programmers are already experienced with.

4.1.1.2 Platform Independence

Applications written for the virtual machine should be able to run on any platform (provided that a virtual machine implementation is available for the platform). WSNs may consist of different sensor node architectures. When an application update is required it would be beneficial to be able to update all different architectures with the same update so as to minimise data transmitted over-the-air. Therefore, platform dependent solutions described in Chapter 2 such as Way Ahead of Time compilation does not suit this requirement. A stack based bytecode instruction set, such as Java bytecode, is a perfect candidate to achieve this since no assumptions are made on to the underlying hardware register set (Shi et al., 2008).

4.1.1.3 Memory Efficiency

Memory on typical sensor nodes is highly restricted (typically from 4 to 16 KB) and therefore the virtual machine should ensure that memory is not wasted and that the majority of the memory is reserved for the actual application rather than the virtual machine.

4.1.1.4 Small Footprint

Like RAM, program space on such resource constrained devices is restricted (typically tens of kilobytes). Thus, the virtual machine should be as small as possible to allow a high majority of the program memory to be used by applications. This is why it has been decided to implement a simple AOT compiler as initial steps to determine the efficiency of such an approach. Future work on generating optimized native code may be conducted to analyse the trade-off of an increasing virtual machine size.

4.1.1.5 Driver Extensibility

Most virtual machine approaches hide the underlying hardware and devices from the application developer. Although the intentions of this are good, since the developer

does not have to worry about the underlying hardware and devices, by not providing a mechanism to interact directly with the hardware it is impossible for an application developer to integrate new hardware features or devices. Thus, the developer is heavily dependent on the virtual machine implementation to provide such abstractions. Such a dependency would preferably be avoided and therefore allow developers to communicate directly with the underlying hardware if required.

4.1.1.6 Software Management

Sensor networks, like other computing platforms are prone to software updates due to various reasons including bug fixes, new features or a complete new application. For this reason the virtual machine must incorporate a mechanism to load and unload software at run-time. For this reason, a software management module has been included in the virtual machine which enables adding or removing software at the granularity of classes or functions.

4.1.1.7 Low-cost

The successful adoption of WSN technology is highly dependent upon lost cost solutions. Motivation behind the work is to lower application development costs. Hardware costs must also be low in order to provide a viable solution. Until low power Java processors become cheap enough and sensor nodes are made available with them on it, using a Java processor may increase costs. Although this area of work is interesting and may provide a viable solution in the future, as it currently stands today the production of such sensor nodes is drastically low. Therefore, we have opted for a solution that will run on top of widely available sensor nodes.

4.1.1.8 Execution Efficiency

Last but not least, a virtual machine should incorporate an efficient execution environment. Although it is true that typical sensor node applications involve a high amount of sleeping, this does not justify high execution overheads as can be seen from the previous analysis. Thus, if it is possible to implement a sufficiently more efficient execution platform, then it is definitely an area which should be investigated. This work proposes AOT compilation of bytecode in resource constrained devices which is widely assumed as impossible or impractical given the resource constraints (Palmer, 2004; Koshy and Pandey, 2005; Pandey and Koshy, 2006; Koshy et al., 2008; Aslam, 2011).

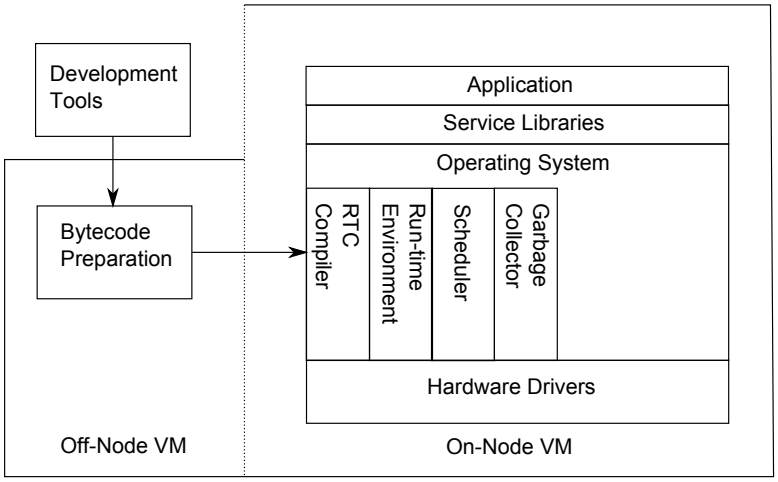

Figure 4.1: This figure depicts the run-time compilation split virtual machine model. The application is developed, compiled to bytecode and then requires to be disseminated into the sensor network. Bytecode is prepared off-node prior to dissemination into the network and is considered part of the same virtual machine abstraction. Received code is loaded by the run-time compiler.

4.1.2 System Architecture

The proposed system architecture is depicted in Figure 4.1. Applications are developed and compiled to bytecode. Similar to the Squawk Split VM architecture proposed by Simon et al. (2006), bytecode is prepared off-node prior to dissemination into the WSN (for several reasons which will be described later in this chapter). This off-node preparation phase is considered part of the abstract run-time compilation split virtual machine model. Once code is received by the run-time compiler it is loaded into the system in the on-node VM. To facilitate the work proposed in this thesis a standard application stack was required to support the execution of applications which typically includes hardware drivers (for devices such as the radio), the operating system components, service libraries (such as a MAC layer), and an application which is loaded into the system. Operating System components required includes the run-time environment which facilitates the execution of code (this includes stack management, amongst other components described later), a scheduler which supports threading, and a garbage collection mechanism to free memory no longer being used by the application. The off-node bytecode preparation phase implies a similar model to the Split VM model. This is required since loading of code often involves extensive memory usage and also the bytecode format generated from the Java compiler is not suitable for such devices. This is especially true when targeting resource constrained devices such as those typically used in WSNs. A more detailed description of this preparation phase, by the Converter component, is discussed later in this chapter.

It was decided to use Java as the candidate programming language so as to have a fair comparison with other proposed virtual machines for WSNs. This results in not only

using Java as the front-end programming language, but also Java bytecode compiled from the standard Java compiler. Other proposed VMs also use a Split VM architecture whereby Java compiled bytecode is further compacted and altered to a bytecode representation more suitable for the targeted devices. Therefore, a similar approach was also taken to allow for a fair comparison. However, supporting a Java bytecode encoding with alterations to further support a more compact and suitable encoding for WSNs may not be the ideal approach. As described in Chapter 2, Java bytecode was not designed for 8 or 16 bit microcontrollers. This is obvious from the fact that operations for 8 and 16 bit datatypes are not inherently supported. To support such operations casting from a 32 bit datatype is required (which results in higher memory usage and computation). This is in fact the reason why such alteration to the Java bytecode generated from the standard Java compiler is required (as will be discussed later). A bytecode encoding which inherently supports 8 and 16 bit datatypes, such as .NET's Common Language Infrastructure (CIL) (Miller and Ragsdale, 2003) or the LLVM assembly language (Lattner and Adve, 2004), would be better suited. However, as previously mentioned a Java bytecode encoding was sustained to allow for a fair comparison with other proposed approaches.

4.1.3 Compilation Process

The compilation process involves the off-node bytecode preparation stage and on-node run-time compilation. A four stage compilation process is proposed, as depicted in Figure 4.2. In the first stage pre-processing is performed which includes resolving microcontroller specific register symbols to a special register array that has been defined which is used for direct access to microcontroller registers (access to registers is explained in Section 4.1.5). The second stage involves compiling 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 a more compact intermediate bytecode. This process involves resolving constants, class names and function signatures to a smaller numerical representation as described later in this section. All external class and function references are resolved against an archive of bytecode translation tables which store 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 AOT compiler.

4.1.3.1 Java Source Pre-processor

The pre-processor is responsible for translating any symbols used into source which can be compiled by the Java compiler. This includes translating microcontroller register

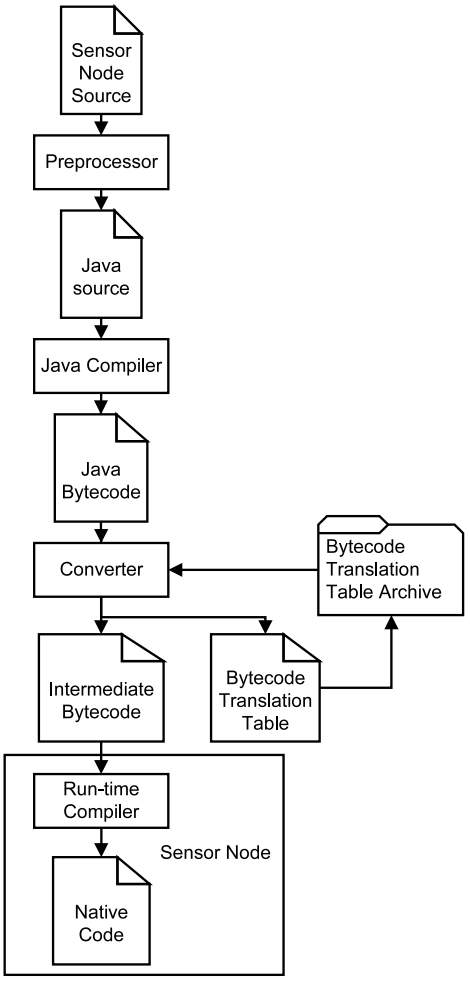

Figure 4.2: The translation process from Java source code down to native code compiled on the sensor node.

symbols into accesses to a special array that has been defined for microcontroller register access described later in Section 4.1.5.

4.1.3.2 Converter

Java was not initially intended for such resource constrained devices, and thus Java bytecode imposes certain properties which are not desirable for such resource constrained systems. The converter is responsible for translating the generated Java bytecode to an intermediate bytecode which is designed around the resource constraints inherent in such devices. Other JVM proposals for WSNs demonstrate substantial offline bytecode optimisation. The only offline bytecode optimisation used in this work has been to remove the constant pool and constant pool lookups. The optimisations considered in other approaches are applicable for this work and should be included in the future to increase performance.

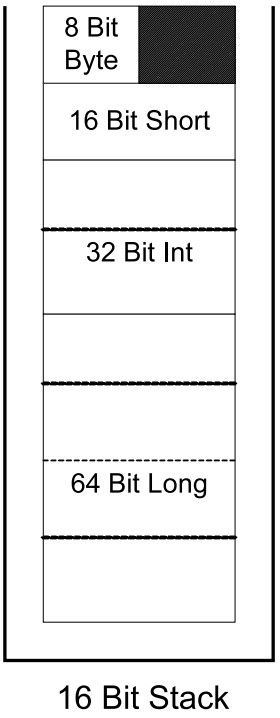

Figure 4.3: A 16 bit operand stack depicting usage of 8, 16, 32 and 64 bit datatypes.

Java uses a 32 bit width operand stack, i.e. objects placed on the stack take up multiple slots of 32 bits. WSN applications however primarily use 16 bit mostly and thus a large amount of memory will be wasted on the stack. Therefore, Brouwers et al. (2008a) proposed to use a 16 bit width operand stack as originally demonstrated by Lindholm and Yellin (2005). Figure 4.3 depicts the usage of a 16 bit operand stack with 8, 16, 32 and 64 bit values placed on top of it. In comparison to a 32 bit stack as depicted in Figure 2.3, less memory is wasted, since 16 bit values do not incur any memory wastage. Translation from a 32 to a 16 bit stack occurs during the bytecode pre-processing stage. Aslam (2011) proposes a variable size slot whereby the developer can choose an 8, 16 or 32 bit width operand stack. Thereafter, in their results they show that minimal memory is freed by using an 8 bit stack for the benchmark applications and therefore the authors propose that a 16 bit width stack is sufficient. In order to support a variable size slot, the virtual machine footprint is increased, and therefore it is questionable whether a variable size slot will provide any benefit for WSN applications and their target platform due to the increased footprint.

The smallest value that the Java bytecode specification (Lindholm and Yellin, 1999) supports is 32 bits. Therefore, in order to translate from a 32 to a 16 bit width stack, changes would need to be made to the bytecode produced for 8 and 16 bit values and variables. Any values that can be stored in 16 bits are translated to 16 bit versions of the respective bytecode instructions. However, to correctly determine whether bytecode instructions can be converted to 16 bit equivalents, analysis on the bytecode must be

performed to ensure that the conversion does not lose any bits that are relevant. In order to achieve this Brouwers et al. (2008a) describe arithmetic optimisation which translates arithmetic operations originally encoded as 32 bit operations to 16 bit operations. Also, since 16 bit bytecode operations have been introduced it could be the case that a 16 bit value will reside on the stack when in fact a 32 bit integer is required. In this case a type cast instruction, `s2i`, can be used to instruct the VM to widen the 16 bit value to 32 bits. The converter presented here also converts the 32 bit stack width generated from the standard Java compiler to a 16 bit stack as proposed in the previous JVMs for WSNs.

When a constant value is used within Java bytecode, a constant value lookup is required. An instruction which uses a constant will specify a 16 bit constant pool lookup index by which the constant will be identified in the constant pool. This lookup although trivial involves an extra lookup phase. Also, the lookup index consists of 16 bits and since most values used in resource constrained development are of the size of 16 bits (or less), then the constant value could be directly placed inline with the instruction. The converter also removes the constant pool by placing constants inline with bytecode commands rather than requiring a constant pool lookup.

Similarly, Java method invocation bytecode instructions require a constant pool lookup index to be specified following the bytecode command. The constant pool lookup index will point to a symbolic reference (in the form of a fully qualified Java method name) to the respective method. Textual symbolic references have little use inside a wireless sensor network, and therefore they are completely removed by the converter which creates a more compact encoding. The textual representations, although helpful when looking at bytecode, substantially increases the size of Java classes. It was therefore decided to replace all class names with a byte value which is unique to that class for a specific wireless sensor network application. This limits the maximum number of classes which can be used in a system to 256, however this can easily be increased to a 16 bit identifier. The same applies to functions, i.e. all functions are resolved to a byte which is unique within the same class, thus restricting the number of functions within a class to 256, resulting in a maximum of 65,536 functions in the whole virtual machine. This maximum limit is sufficient since many resource constrained devices have a 16 bit program address space. However as previously mentioned it is an easy task to increase this maximum value.

As an example consider the invocation of a static method as depicted in Figure 4.4. The sequence on top represents standard Java bytecode. The standard Java `invokestatic` bytecode instruction requires a 16 bit constant pool lookup, in this case index #2. The virtual machine is then required to perform a lookup in the constant pool for item #2. The constant pool item will contain a reference to the method requiring invocation, which can then be invoked. The converter removes the constant pool by directly appending constant values following instructions. In the case of invocation the class and method

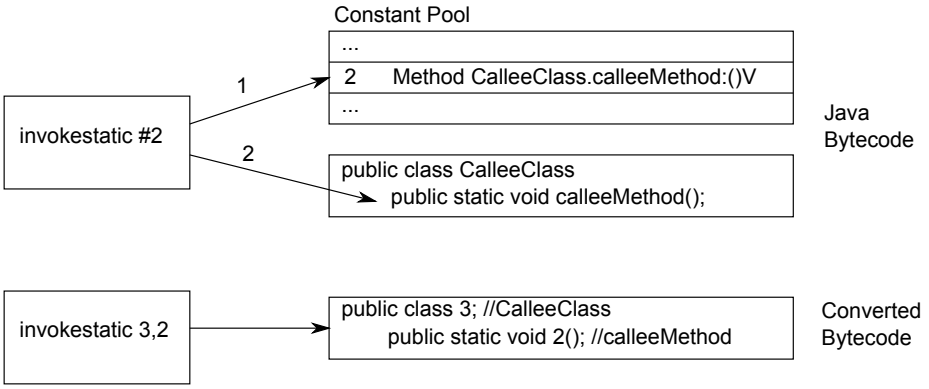

Figure 4.4: Bytecode method invocation execution for standard Java bytecode and the proposed converted bytecode.

Table 4.1: Bytecode to Native Code Translation Example

Bytecode	Stack Before	Stack After	Pseudo Native Code
iload_1	...	..., value1	PUSH variable1
iload_2	..., value1	..., value1, value2	PUSH variable2
iadd	..., value1, value2 ..., value1	..., value1 ... ..., result	POP reg1 POP reg2 ADD reg1, reg2, reg3 PUSH reg3
istore_0	..., result	...	POP variable0

numeric identifiers are appended to the invocation bytecode instruction. Therefore, the method to be invoked can be directly referenced rather than having to lookup the value in the constant pool. As described above 8 bits is sufficient to encode both the class and method numeric representations. Thus, the method can be directly referenced without changing the parameter size of the invocation instruction, which will result in a more efficient execution of invocation instructions. More so, this facilitates the removal of the constant pool, which will reduce bytecode size for applications utilising a majority of 8 and 16 bit constant values.

4.1.4 AOT Compiler and Execution

The AOT 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 AOT compiler, a run-time operand stack was designed which mimics the Java bytecode operand stack in a similar fashion to the baseline compiler proposed by Suganuma et al. (2000). It is believed that a substantial decrease in execution overhead can be achieved without introducing any complex compiler optimizations, since it is planned to remove the high interpretation overhead as previously described in Chapter 3.

The AOT compiler will perform a single pass on the bytecode which it is required to load. The bytecode to native code translation process is done once per unit of code. When a unit of code is translated to native code it is stored in flash memory for future execution. The initial AOT compilation process occurs as bytecode is sent to the device. Thereby the bytecode is never stored in the device and is immediately translated to native code. That said the installation process is similar to the installation of bytecode on an interpreter. The main difference is that the AOT compiler will require to translate the bytecode to native code, and then save the native code to flash (which is larger than bytecode). The overheads in doing so are minimal, especially if this process is considered to be at the bootloading stage when sensor nodes are likely to be externally powered.

The AOT compiler follows a set of rules that define how each bytecode instruction will be transformed to native code. The general ideology behind the translation, as stated above, is that the native code will perform actions so that at every bytecode instruction the operand stack will mimic the contents of the Java operand stack (even if such actions can be removed). Table 4.1 provides an example of a step by step execution of native code for the expression c = a + b. The stack's contents before and after execution of the native code instructions.

4.1.4.1 Double-Ended Operand Stack and Garbage Collection

The Java operand stack is used to push and pop integer and reference operand values to which will be used by operations. Garbage collection requires to determine whether references to objects exist within the VM in aim of identifying memory no longer being used. The garbage collector therefore, amongst other things, requires to iterate through the operand stack to detect if any references exist on the stack. Operands placed on the stack are typically marked up with a type identifier used to associate the operand to a specific data type or class. In order to facilitate a more efficient garbage collection scheme, Brouwers et al. (2009) proposed to separate references from integer operands on the stack. Thereby type information can be removed from the stack and also the garbage collector will only require to traverse the references placed on the reference stack (without having to determine the types of the operands). To implement this Brouwers et al. (2009) propose that the integer values should be pushed on one end of the stack and reference values pushed onto the other end of the stack. A depiction of how the stack is separated into a double-ended stack is shown in Figure 4.5. To facilitate garbage collection it was also decided to implement a double-ended operand stack in the work proposed in this thesis. Similar to separating references from values on the operand stack, the same is done for object fields. Each object separates the value fields from reference fields by keeping separate pointers to the values and references associated with the object. Thereby, the garbage collection scheme will not require to check whether slots on the stack or fields are references or values.

Figure 4.5: Double ended stack depicting reference slots pushed on to the left of the stack and integer slots pushed on to the right of the stack.

```
01    function mark(Object object)
02      if object.markbit == 1
03        return;
04      object.markbit = 1;
05      for each (Object fieldObject in field_objects)
06        mark(fieldObject);
07    end function
08
09    function gc()
10      clearMarkBits();
11
12      for each (Object object in static_object_list)
13        mark(object);
14
15      for each (MethodFrame methodFrame in method_frames)
16        for each (Object object in methodFrame.referenceStack)
17          mark(object);
18        for each (Object object in methodFrame.referenceVariables)
19          mark(object);
20
21      sweepUnreachableObjects();
22    end function
```

Figure 4.6: Garbage collection pseudo code.

In this work a naïve mark and sweep garbage collector was implemented. Each object has an associated bit which represents the mark bit. The bit is used to identify whether the object is reachable from any root execution points. This involves three stages: clearing the mark bit for all objects on the heap, marking all objects that are reachable, and removing any object from the heap that after marking have been found to be unreachable. The process of clearing the mark bit is straight forward. The heap is traversed and the mark bit is cleared for every object on the heap. Setting the mark bit for all reachable objects is only slightly more complicated. Pseudo code describing the procedure is provided in Figure 4.6. The `mark` function is used to mark an object as being reachable. An object may also reference other objects, and therefore each object referenced by the object being marked is also set to be marked in a recursive manner. The `gc` function represents the garbage collection main function which will first clear all mark bits for all objects on the heap and finally after marking each reachable object, it will sweep all unreachable objects represented by `clearMarkBits` and `sweepUnreachableObjects` respectively. The pseudo code for the clear bits and sweep functions have not been

included due to brevity and also the functions' simplicity. The mark phase involves two parts: marking all objects that are static objects (and inherently all recursively referenced objects), followed by marking all objects that are on any reference stacks (and again all recursively referenced objects). The list of reference stacks includes reference stacks for all functions on the call stack for every possible thread of execution.

If the operand stack was not separated into reference and value operand stacks, and similarly if class fields and method variables were not separated into value fields and object fields then a check would be required on each field and operand to determine whether or not they are in fact objects. Therefore, lines 5 and 6 in Figure 4.6 would be altered to loop through every possible field (instead of only object fields), and also for each field a conditional statement would be required to determine if the field is an object. The same applies for lines 16 and 17, in that all operands on the stack would require to be traversed (instead of just the reference operand stack) and the type of each operand would be required to be checked to determine if it is an object. Variables are also split up into value variables and reference variables and therefore the same would need to be applied to lines 18 and 19. Therefore, the complexity of these two loops, in Big O notation, have been reduced to $O(r)$ from $O(r + v)$ where r is the number of reference objects and v is the number of value variables or fields in the associated field list or operand stack.

4.1.4.2 Stack Frame Layout

The traditional JVM stack scheme allocates a single JVM stack per thread. A method stack frame is created and placed on the JVM stack upon method invocation. This involves allocating enough space for local variables and necessary bookkeeping information. The operand stack is placed on top of the local variables and bookkeeping information, which grows and shrinks as values are pushed and popped onto the operand stack. A traditional JVM stack is depicted on the left of Figure 4.7. Arguments intended for a callee function are pushed on the caller's operand stack. Thereafter the same arguments in memory can be used as the starting point for the local variable section of the callee function. Thus, the argument values only exist in one memory location. This scheme minimises memory consumption by not having to copy arguments over to the new method. However, the drawback of this scheme is that the JVM stack size has to be pre-allocated on thread initialisation in order to provide a contiguous memory section for the JVM stack. This means that the worst case JVM stack size must be pre-allocated even if not fully in use (which will be the case throughout most of the execution of a program). Memory on WSN devices is severely limited and thus this scheme may not be appropriate.

Brouwers et al. (2009) propose a different method which uses linked stacks (von Behren et al., 2003). Linked stacks provide a scheme whereby a method stack frame is allocated

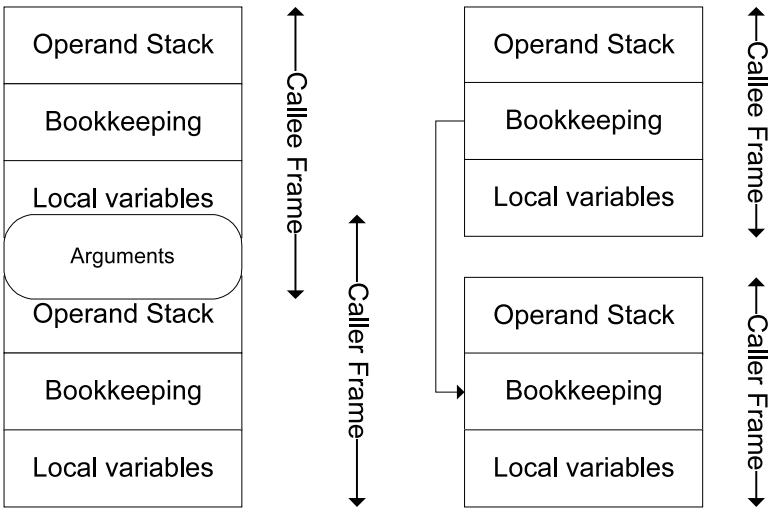

Figure 4.7: A traditional JVM stack depicted on the lift and linked stacks on the right.

upon method entry and freed upon method exit. Figure 4.7 depicts a traditional JVM stack scheme on the left and a linked stack scheme on the right. The benefit of using linked stacks is that memory related to method stack frames are only allocated for those methods that are executing. The drawback on the other hand is that, since the method stack frames do not share a contiguous memory layout arguments from one function to another must be copied over from the caller's operand stack to the callee's local variables. However, given the target platform, this slight duplication of arguments may justify the decrease of pre-allocated memory required for a thread's JVM stack.

The traditional JVM stack scheme frame layout minimises memory consumption since argument values are not required to be copied from caller to callee stacks, however it requires that the whole stack frame is preallocated on a per thread basis. Although memory consumption is reduced for argument value passing the benefits are only heeded when the entire stack frame size is used. This will only occur very rarely in a program's life cycle. More so, the worst case size of the stack frame is not known and usually a substantial amount of memory is allocated for it. Given the limited amount of memory available on the targeted device it would not make sense to preallocate substantial memory for the JVM stack layout especially when most of the time the JVM stack will be quite empty. Linked stacks on the other hand allow for memory to be allocated to stack frames as required, however at the cost of duplicating argument values copied from caller to callee functions. Linked stacks require memory allocation to take place every time a function call is made, whilst the the traditional stack layout only requires an initial stack frame layout allocation. Therefore, linked stacks will incur a memory allocation execution overhead every time a function is called.

The memory allocated for each linked stack frame is automatically deallocated when the associated function exits without requiring garbage collection. However, since more

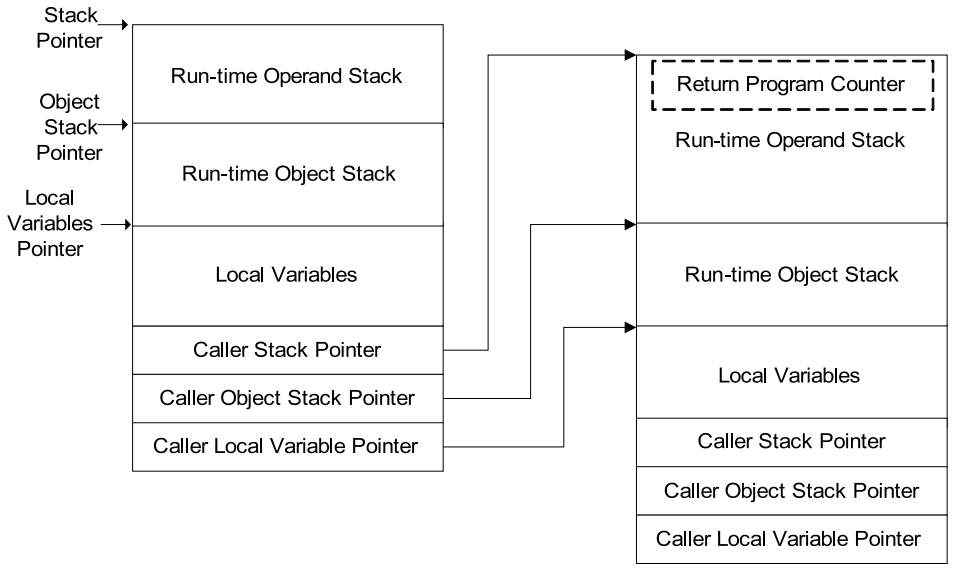

Figure 4.8: Run-time linked stack frame layout.

memory is required for linked stacks due to duplicated argument values, garbage collection may be required more frequently since less memory will be available to the application. That said, a traditional JVM stack frame layout will most of the time result in more garbage collection cycles since the complete stack frame layout memory will be allocated throughout the execution cycle of the application, resulting in less memory availability during periods when the whole stack frame layout is not required. Linked stacks may cause system failure if not enough memory is available to the application to cater for application memory, the stack frame memory and the duplicated argument memory. A traditional JVM stack frame layout will allow for more memory to be utilised by the application (since it does not require additional memory such as the duplicated arguments), however due to its statically initialised size, it will be very difficult to find the sweet spot where the JVM stack frame layout size fits perfectly with the worst case memory requirement. Also, linked stacks will be able to cater for dynamic situations whereby more application memory is required than stack frame memory or vice versa; whilst a traditional JVM stack frame layout will not be able to cope with such dynamicity. On traditional computing platforms it is usually sufficient to use a traditional JVM stack frame layout, since ample memory is usually allocated for the JVM stack frame layout and also for the heap. However, the resource constrained devices being targeted do not have that luxury and therefore linked stacks provides a more suitable solution (even given the extra execution overhead and temporal memory overhead).

The run-time linked stack frame layout used is depicted in Figure 4.8. Two stack frames are present; the stack frame on the left belongs to a method which has been called by a method associated with the stack frame on the right. Stack frames contain local variables, the integer operand stack and the object stack (which are implemented as a double ended stack). A pointer to each of the stacks and the local variables are

maintained for the active stack frame. A stack frame also maintains pointers to the caller's stacks and local variables so that the active stack frame pointers can be set to them once the callee method has completed. Before a method calls another method it also pushes the return program counter address onto the stack. Then when the callee changes the current stack frame to the callers stack frame, no other action is required to restore execution to the caller's program counter since it is already on the native stack (the hardware run-time environment then pops the top element to establish where to set the program counter to).

The work presented in this thesis provides methods to enable run-time compilation in severely resource constrained devices which in turn provides a native Java execution environment. That said, the execution environment is independent to other system components such as Garbage Collection and Threading. In order to test the work a full system architecture was required and therefore it was decided to implement a Naïve Mark and Sweep Garbage Collector and pre-emptive threading, however different garbage collection and threading or event driven models could be implemented without affecting the run-time compilation techniques proposed. The pre-emptive threading implementation will be described next.

4.1.4.3 Threading

To demonstrate that the run-time compilation techniques are independent of the execution paradigm, it was decided to implement a pre-emptive threading execution paradigm. An overview of implementation details will now be provided. In traditional JVMs each thread is assigned its own JVM frame stack, however as described, it was decided to opt for linked stacks to minimise memory consumption for majority of the application lifetime. Therefore memory does not need to be preallocated for the thread, however memory will be allocated upon function execution. The thread does, however, require to acquire memory for its process control block. The process control block is used to store the state of a thread's registers so that when a thread regains execution control its last executing state is restored. The thread's execution starting point memory address is stored into the process control block, and thereafter a global thread list is updated to include the thread's object reference and the process control block. Thread deletion involves removing the thread's process control block and object reference from the global threads list, and will also actively force a context switch if the current executing thread is the thread being deleted. Java inherently deletes threads when the thread execution ends. Therefore, thread deletion is implicitly performed at the end of each thread's execution point.

One of the microcontroller's internal timers is used to interrupt execution occasionally and initiate a context switch. The context switch involves storing the current executing thread's registers (including the stack pointer), choosing a new thread to execute (a

round-robin thread execution model has been implemented) and restoring the thread's registers including the stack pointer and program counter. Since linked stacks are used, an initial thread frame stack is not allocated. Linked stacks require memory to be allocated upon method invocation. Therefore no changes need to be made to support threading since memory will be allocated as required when methods are called independent of the execution model. More so, the threading implementation is completely indifferent to garbage collection as well. The reason is that the threading operations are executed as atomic sections (that is they cannot be interrupted) and the same applies for garbage collection.

Java provides two bytecode instructions, `monitorenter` and `monitorexit`, to support thread synchronisation. The instructions provide a locking mechanism in which an exclusive lock can be obtained on an object. `monitorenter`, as specified by the Java Virtual Machine Specification, will obtain a lock on the object and continue executing if the object is not already locked. If it is locked by a different thread, then the executing thread will block and wait until the object is unlocked. If the executing thread requires another lock to an object which it already obtains a lock for than a lock counter will be increased. Therefore, to support this each object is associated with a lock counter and a reference to the owning thread. Blocking is implemented by forcing a context switch. As soon as a blocked thread receives execution back it will again check if the lock has been released, and if not once again block. `monitorexit` is straightforward and only requires to decrement the lock count.

4.1.4.4 Optimizations

Since most JIT compilers run on powerful machines they tend to include algorithms which generate highly optimized code. Also, such compilers do not tend to take the same approach that is taken in this work, in that they do not attempt to maintain a run-time operand stack but instead map the operations directly to registers. Although, the IBM baseline compiler (Suganuma et al., 2000) does use a run-time operand stack, the purpose of the development of the IBM baseline compiler was to serve as initial steps for JIT compilation and to verify the other compilers as they were being developed (Burke et al., 1999). Two types of optimisations are presented here, being optimisations focusing on execution speed improvements (that may also decrease size requirements) and optimisations that focus on decreasing size (which may hinder speed improvements).

The speed optimisations implemented are in the form of peephole optimisations that are performed on the generated native code (which is stored in a buffer that will be described later). The following peephole optimizations are performed on the baseline generated native code:

Table 4.2: Optimization Examples

Before			After		
Instructions	Cycles	Length	Instructions	Cycles	Length
PUSH R13 POP R13	6	2		0	0
PUSH R13 POP R14	6	2	MOV R13,R14	1	1
MOV #0,R15	2	2	CLR R15	2	1
MOV R6,R5 MOV R5,0x0000(R4)	5	3	MOV R6,0x0000(R4)	4	2

- Completely removing PUSH/POP pairs which work on the same register, since the pair has no effect.
- Resolving PUSH/POP pairs working on different registers to a direct register move command.
- Resolving any 0 constant move commands to clear commands.
- Removing any intermediate register copy commands.

Examples of the optimizations are given in Table 4.2. The instructions both before and after optimization are listed along with the number of clock cycles it takes to execute the instructions and the length of the instructions in words. The examples are based on the MSP430 architecture which has a 16 bit word size. The first two optimizations provide both cycle length and instruction size optimizations, whilst the last one provides size optimization only. These optimisations are considered as speed optimisations (although they do decrease size as well).

Optimisations focused on decreasing the size of a native code image are also included into the design of the compiler. A single bytecode instruction could result in more than ten native code instructions. In order to reduce the generated native code size, assembly functions were created that perform the same native code required for bytecode instructions that entail a larger number of native code instructions. Then when such a bytecode instruction is required to be compiled a native call to the associated assembly function implementing the logic will be generated. As an example consider the Java bytecode instruction `iastore` which is used to store an `int` value in an array at a specified index. The non-optimised code generated for the instruction is as follows:

```
POP R11 ;pop the most significant value byte into R11
POP R5 ;pop the least significant value byte into R11
POP R6 ;pop the index into R6

;pop the array reference from the reference stack
MOV.W @R8, R7 ;copy the value on the reference stack to R7
```

```
DECD.W R8 ;decrease the reference stack (R8) by 2

;R7 points to the array, skip array information
INCD.W R7 ;skip the array type
INCD.W R7 ;skip the array length

;array items are 4 bytes; find the byte offset from the index:
RLA.W R6 ;left shift the index a first time
RLA.W R6 ;left shift the index a second time

;point R7 to the array item
ADD.W R6, R7 ;add the byte offset to the start of array data

MOV.W R5,0x0000(R7) ;copy the LSB to the array item LSB
MOV.W R11,0x0002(R7) ;copy the MSB to the array item MSB
```

The generated native code above is 28 bytes. A native call only consists of 4 bytes. Therefore, by increasing the run-time compiler footprint by 28 bytes (plus 6 additional bytes for function setup and returning) the application size requirements for the `iastore` instruction can be decreased to 4 bytes. Therefore, if the `iastore` instruction is used more than 7 times then overall compiler footprint and application size requirements will also be decreased. Other similar size optimisations have also been included however have been left out for brevity. That said, an extra `CALL` native code instruction is introduced along with function preparation and returning from the function which will incur a higher execution overhead. The original generated native code above would also require 28 clock cycles to execute. The additional instructions for the size optimisation would require 5 clock cycles for the call instruction; 6 clock cycles for function preparation and 2 clock cycles to return from the function. That is, the size optimisation would result in an additional 13 clock cycles, an additional 46% execution time. The execution slowdown is inherent in the size optimisation and is a trade-off that needs to be considered when such optimisations are chosen.

4.1.4.5 Gradual Compilation of Bytecode

Typical Java JIT platforms compile whole functions at a time or at the minimum require the function's complete code in order to perform JIT compilation. This would involve storing the said function in memory and then performing compilation once the whole function is received. In aim of decreasing memory overhead when receiving code, this work proposes gradual compilation which allows code to be compiled as it is received rather than waiting for whole functions to be received. This reduces the required memory to perform compilation to that of tens of bytes.

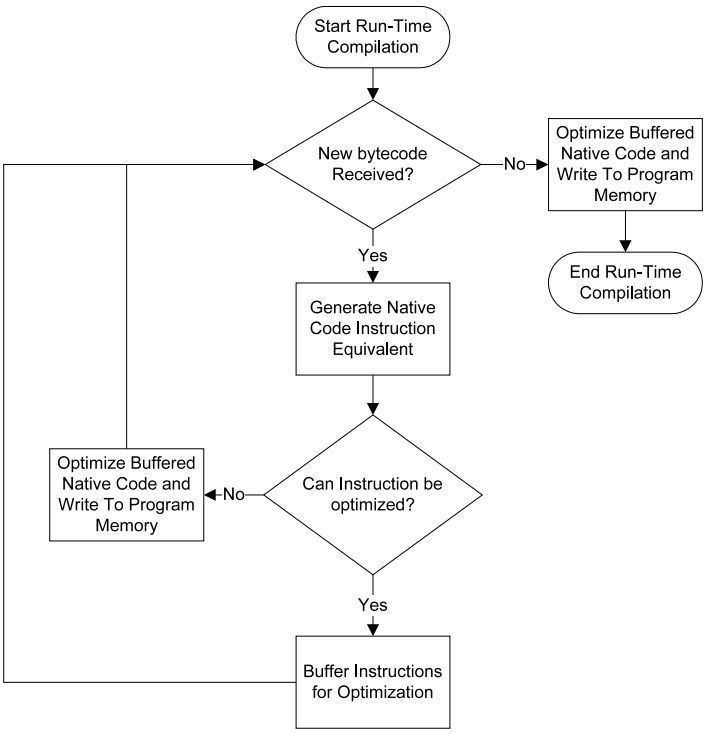

Figure 4.9: The gradual compilation algorithm which buffers instructions that can be optimized and then writes the optimized native code to program memory when instructions which cannot be optimized are received.

The reason why such compilation techniques require a function's complete code listing is due to the requirement to resolve jump destinations. In order to be able to compile code without a function's complete code listing, it has been decided to pass the bytecode jump locations at the beginning of new functions. Thereafter, upon receiving and compiling bytecode to native code, the native code equivalent location of the bytecode jump location can be filled in as it is encountered and any code which jumps to this location can be patched with the native code equivalent location.

The AOT compiler mimics the Java operand stack by natively pushing and popping operands to the microcontroller's stack. `PUSH-POP` sequences are reduced to `MOV` instructions or completely removed, therefore the gradual compilation process has been constrained to buffer the generated native code and only write the buffered native code to program memory when non-optimizable instructions are encountered. The gradual compilation algorithm is presented in Figure 4.9. Upon run-time compilation and whilst new bytecode is received, any generated instructions that can be optimized are buffered. When an instruction is encountered that cannot be optimized, all instructions in the buffer are optimized if possible, and then written to program memory. When the run-time compiler is no longer awaiting instructions it will again optimize any instructions in the buffer if possible, and thereafter write the instructions native code to program memory.

4.1.5 Hardware Register Access

As outlined in the requirements, it would be beneficial to expose the underlying hardware to the application programmer in aim of facilitating hardware driver development and to provide access to the microcontroller's features if required. Common microcontrollers interact with their various peripherals by means of a register set that can usually be accessed as part of the data address space. It has been decided to expose this to the application developer as an array which can be read from or written to which is later translated to read and write operations on the actual data address space. To facilitate hardware register access even more, the microcontroller's symbol table has been added to the pre-processor. The translation process of microcontroller symbols is shown in Figure 4.10. Upon seeing any of the microcontroller's symbols, the pre-processor will translate them to array accesses via the fully qualified Java namespace for the relevant register. The Java compiler and converter will generate bytecode which is then passed into the run-time compiler which compiles it to native code.

4.1.6 Exposing Interrupts

Microcontrollers raise interrupts for different events that can occur. In fact, implementing code in interrupts can help minimize power consumption since modern microcontrollers can sleep the rest of the time and only wake up when an interrupt is required. An application developer working on a virtual machine should not be restricted to what

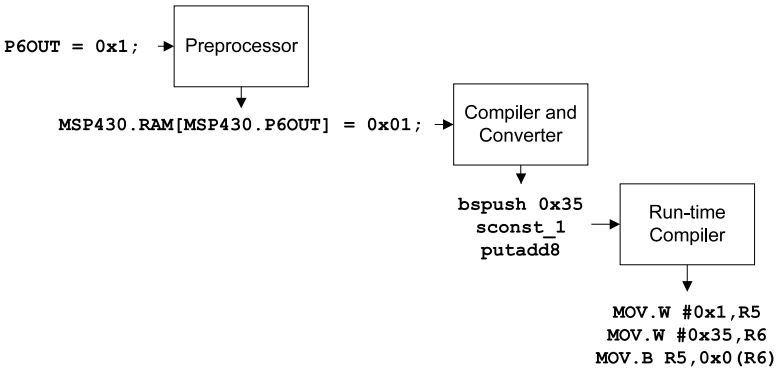

Figure 4.10: Translation process of microcontroller symbols.

```
@InterruptAttribute(interrupt=TIMERA0_INTERRUPT)
public void TestInterrupt()
{

}
```

Figure 4.11: Example of how an interrupt routine is exposed to developers.

has been exposed by the virtual machine, but if need be, the developer should be able to extend such functionality. Thus, it has been decided to expose interrupts to the application developer via a custom Java attribute class created, named `InterruptAttribute`. The developer will only be required to annotate the method which is to serve as the interrupt with the attribute `InterruptAttribte` and the associated interrupt identifier. Figure 8.2 is an example of code that can be used to create an interrupt, in the case of the example it exposes the `TIMERA0` interrupt.

4.2 Implementation Specific Details

An Ahead-Of-Time compiler for the MSP430F1611 microcontroller has been implemented and has been tested on the Telos B (Polastre et al., 2005), TinyNode 584 (Dubois-Ferriere et al., 2006) and BSN (Lo et al., 2005) sensor nodes. As discussed in the Design section, 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 AOT compiler.

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

Received bytecode is passed into the AOT compiler which converts the bytecode to native code, as described in Section 4.1.4.5. Table 4.3 displays different bytecode instructions and the respective generated native code. 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 push and pop operations an interpreter would execute. The primary difference is that the push and pop operations generated by the AOT compiler do not incur any interpretation overheads. More so, all interpretation overhead has been removed from the execution paradigm.

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

```
iload_1
iload_2
iadd
istore_0
```

Table 4.3: Bytecode to Native Code Conversions

Bytecode	Native Code
iload_0	PUSH <OFFSET>
iconst_0	PUSH #0
aload_0	PUSH <OFFSET>
getfield	POP.W R9 MOV.W @R9,R9 PUSH <OFFSET>(R9)
istore_0	POP R10 MOV.W R10,<OFFSET>(R11)
iinc	ADD.W <CONST>,<OFFSET>(R11)
iadd	POP R10 POP R9 ADD.W R10,R9 PUSH R9
if_icmpeq	POP R9 POP R10 CMP.W R9,R10 JEQ (<JUMP ADDR>)
dup2	PUSH 0x0004(SP) PUSH 0x0004(SP)
putstatic	POP R10 MOV.W R10, <STATIC VAR ADDR>

Although the variables were specified as `short`, the resultant bytecode converts them to `integer`, since Java uses a 32 bit width stack. As previously explained this work makes use of a 16 bit width stack. Thus, the converter will translate such statements to a 16 bit width version. The following intermediate bytecode is produced from the above bytecode by the converter:

```
    sload_1
    sload_2
    sadd
    sstore_0
```

The AOT compiler will then produce the following native code commands for the above generated bytecode:

```
    ; sload_1
0   PUSH <OFFSET TO VARIABLE 1>

    ; sload_2
1   PUSH <OFFSET TO VARIABLE 2>

    ; sadd
2   POP R5
```

```
3    POP R6
4    ADD.W R5,R6
5    PUSH R6

     ; sstore_0
6    POP R5
7    MOV.W R5,<OFFSET TO VARIABLE 0>
```

The `sload_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 `sadd` bytecode instruction results in two native code *poppings* into registers 5 and 6 which are then added together and the result stored in register 6 by the `ADD.W` native command. In order to mimic the bytecode operand stack, the result has to be put on the run-time operand stack. Hence the result stored in register 6 is pushed 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 5 and then moving the value of register 5 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. When using the optimizations described in 4.1.4.4, the generated native code is reduced to:

```
; sload_1
MOV.W   <OFFSET TO VARIABLE 1>,R6
; sload_2
MOV.W   <OFFSET TO VARIABLE 2>,R5

; sadd
ADD.W   R5,R6

; sstore_0
MOV.W   R6,0x0000(R4)
```

In this chapter, techniques to enable AOT compilation for resource constrained devices including a run-time operand stack that mimics the Java operand stack, and gradual compilation. An evaluation of the proposed techniques will now be provided to determine exactly what gains are achieved by using an AOT compiler, and also to establish whether mimicking the bytecode operand stack is a good idea.

Chapter 5

Evaluation of the AOT Compiler

In order to evaluate the Ahead-Of-Time compilation approach proposed, benchmark tests as used by Brouwers et al. (2009) and Aslam et al. (2010) have been implemented. The benchmarks have been implemented using the same Java code source from the Darjeeling and TakaTuka distributions, with only minor changes to remove Darjeeling and TakaTuka specific function calls. Both 16 and 32 bit bubble sort tests, an MD5 test and a binary search test have been tested.

The bubble sort test sorts 256 integer values which are initialised in descending order for 16 and 32 bit integer types. The MD5 test performs 5 MD5 hashes on each of the strings 'a', 'abc', 'darjeeling' and 'message digest'. The binary search test performs 1,000 binary searches for the worst case search in 100 16 bit values.

Only the results for a single experiment for each benchmark were required since the execution of code runs directly on the microcontroller without disturbance or any required input from external events. Execution timings were measured by switching a LED on temporarily and then switching it off before the test, and then switching a LED on after the test. By measuring the current consumption on an oscilloscope, the beginning of the experiment can be pinpointed from the drop in current consumption when the LED is turned off, and similarly the end of the experiment can be marked from the increase in current consumption when the LED is then turned on. The accuracy of the oscilloscope for the experimental setup was 1 ms.

The AOT compiler can be configured for speed and size optimizations. The above tests have been performed for the following configurations: AOT compilation with no optimizations (AOT), AOT compilation with speed optimizations (AOT-Sp), AOT compilation with size optimizations (AOT-Si) and AOT compilation with speed and size optimizations (AOT-SpSi). The size and speed optimisations are those optimisations described in the previous chapter. Speed optimisations involve peephole optimisation of the generated native code whilst size optimisations focus on decreasing size requirements by implementing the required native code in the run-time compiler footprint, and

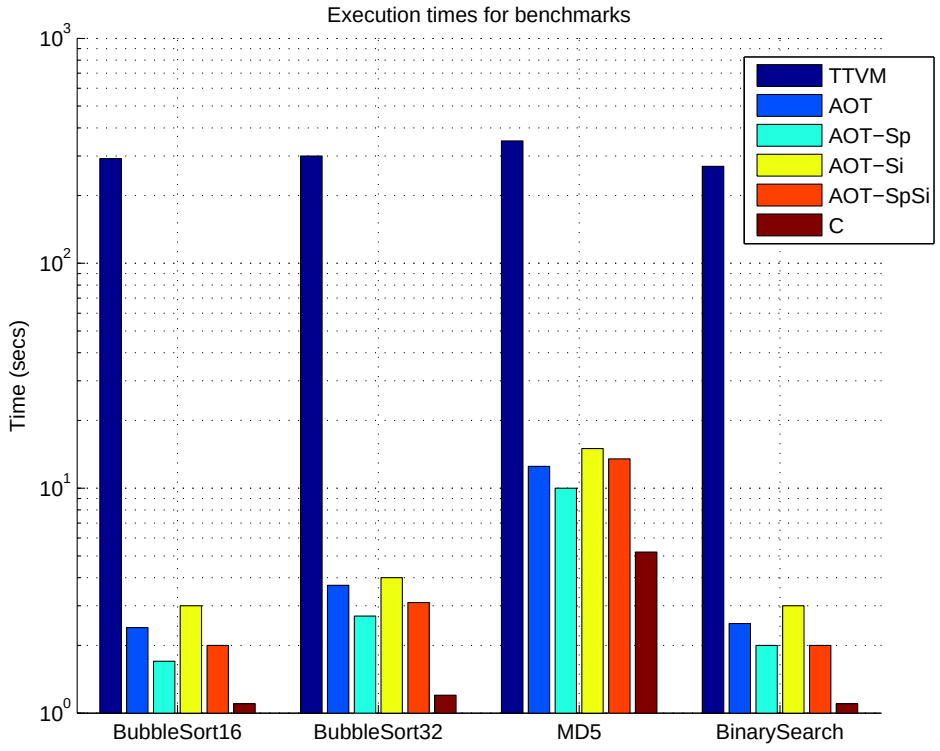

Figure 5.1: Execution times for C, AOT Compiled and TakaTuka versions of the benchmarks. The y axis uses a log scale to better visualise the large differences.

thereafter only requiring a call instruction to the respective instruction code. The same tests were performed on the TakaTuka virtual machine (TTVM) on the same hardware. A Telos B (Polastre et al., 2005) sensor node was used (equipped with a MSP430F1611 microcontroller) running at 4MHz for the above tests.

Unfortunately, the Darjeeling virtual machine is no longer supported for the MSP430 microcontroller and thus could not directly be benchmarked against.

The execution performance of the benchmarks for C generated native code, Interpreted bytecode (using TakaTuka) and the AOT compiler (for each optimization configuration) will be evaluated. Following the execution performance evaluation, an analysis of the generated program size will be provided.

5.1 Execution Performance Evaluation

In order to measure the execution time of the benchmark applications the current consumption of the sensor node was monitored and a LED was used to indicate the start and end of the test.

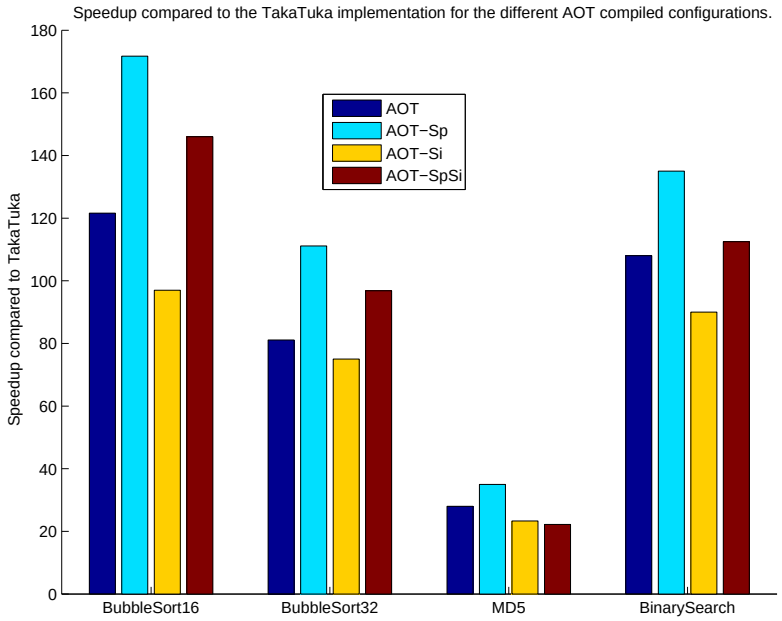

Figure 5.2: Comparison of the AOT compiled benchmarks to the TakaTuka versions of the benchmarks.

The time taken for each benchmark application using the TakaTuka VM, the different configurations of the AOT compiler and C implementations are displayed in Figure 5.1. As can be seen from the graph, AOT compiled code is executed significantly faster than interpreted code. In comparison to native code generated from C implementations, the AOT compiled code executes from 1.8 to 3.4 times slower (whilst the TakaTuka implementations require from 67 to 324 more than that of the C implementations).

Figure 5.2 shows the speedup of the different AOT compiled configurations compared to the TakaTuka VM implementations. The AOT compiled configurations results in varying speedups for the benchmarks ranging between 22 to 171 times the speed of the interpreter.

From the execution speed evaluation of AOT compiled code compared to the interpreter implementation, it is evident that a substantial performance increase is gained. The slowdown incurred for AOT compiled code (1.8 to 3.4 times) compared to the C implementation is due to the stack based architecture which is maintained to simplify the compilation process.

To put these values into perspective, the resultant execution overheads of the interpreter and AOT compiler will be applied into the lifetime model described in Chapter 3. The optimal execution overhead achieved will be used for both the interpreter and the AOT compiled code, i.e. 67 and 1.8 times the C execution time respectively. Thus, 3.4 can be redefined as:

$$T_{int} = T_{active} \cdot 67 \tag{5.1}$$

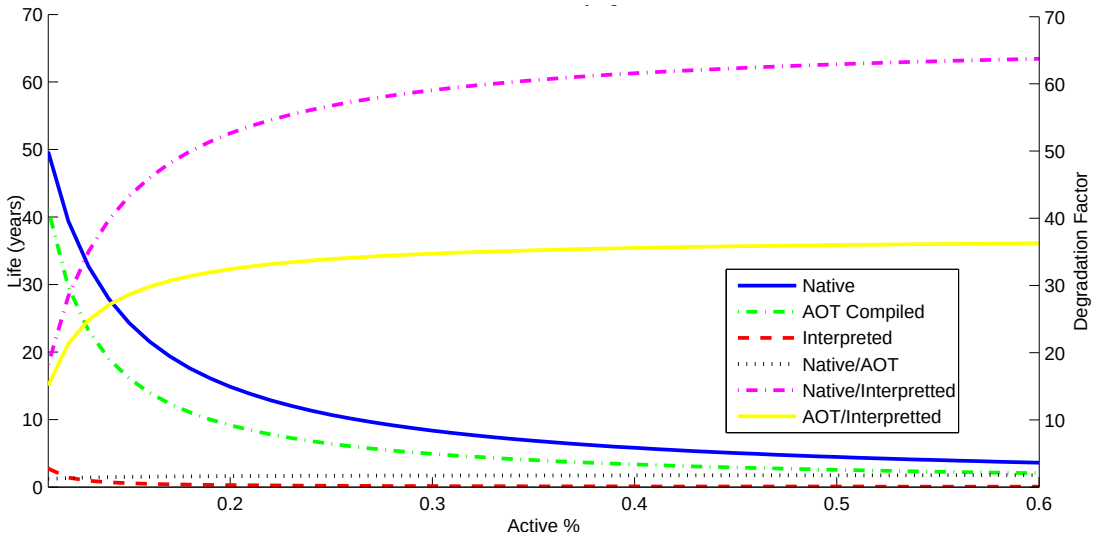

Figure 5.3: Lifetime of a sensor node with a varying active execution ratio for native code, AOT compiled code and interpreted code. The ratio of the lifetime of native code to AOT code and interpreted platforms is also depicted. As the active percent increases the ratio of native code to interpreted lifetimes tends to the overhead of the approach, i.e. 1.8 for AOT compiled code and 67 for interpreted code.

The ratio of time in which the microcontroller is in an active state for AOT compiled code, T_{aot}, can be defined as:

$$T_{aot} = T_{active} \cdot 1.8 \tag{5.2}$$

and L_{aot}, the lifetime of a node equipped with an AOT compiler as:

$$L_{aot} = \frac{B}{P_{active} \cdot T_{aot} + P_{sleep} \cdot (1 - T_{aot})} \tag{5.3}$$

In Figure 5.3, the expected lifetime for a native code implementation is then plotted using equation 3.3, AOT compiled code (equation 5.3), interpreted code (equation 3.5) (using the new overhead value 5.1), and ratios for the expected life of native versus AOT compiled code, native versus interpreted code and AOT compiled against the interpreted code. As the duty cycle increases the native code to AOT compiled lifetime ratio tends towards 1.8 (the AOT compiled implementation overhead) and likewise the native code to interpreted lifetime ratio tends towards 67 (the interpreter overhead). Also, as the duty cycle increases, the AOT compiled to interpreted lifetime ratio tends towards 37, i.e. the overhead of using an interpreter in comparison to AOT compiled code.

Due to Java class abstractions and the stack based architecture of Java and the fact that the AOT compiler sustains the Java operand stack (in order to facilitate a simple small footprint AOT compiler), the efficiency of the AOT compiled code cannot match that of native code compiled from C. C++ compiles out class abstractions by statically linking function calls which produces faster native code. Such an approach could not be

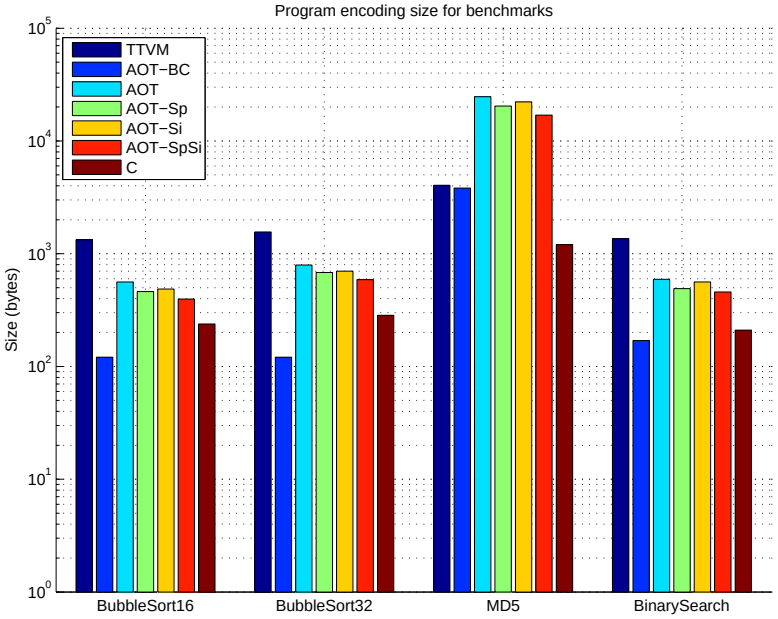

Figure 5.4: The size of the benchmark application logic for TakaTuka bytecode (TTVM), native code for a C based equivalent, and the intermediate bytecode (AOT-BC), the generated AOT compiler native code size for no optimizations (AOT), speed optimizations (AOT-Sp), size optimizations (AOT-Si) and both speed and size optimizations (AOT-SpSi).

taken in this work since the execution of Java methods are dependent on the underlying object type which would be lost when statically linking functions. In any case, the increase in execution overhead is negligible when compared with that of an interpreter based approach. The slight increase in execution overhead is justified by the platform independence achieved by using a bytecode encoding and the higher level programming abstraction provided to developers. Bytecode also usually results in energy gains when transmitting software over the air due to the smaller encoding when compared with native code.

5.2 Program Size Evaluation

Besides the limited battery life and execution power, sensor networks also have a limited amount of program space. An evaluation of the program space required for the different program encodings will now be provided, followed on by an analysis of the size of the virtual machines coupled with the applications.

5.2.1 Program Encoding Size Evaluation

Figure 5.4 depicts the size of the program encoding for TakaTuka bytecode (TTVM), the native code size for a C based equivalent, the intermediate bytecode (AOT-BC),

i.e. the bytecode that is sent to sensor nodes using the AOT compilation technique; as well as the native code generated on the sensor nodes using the AOT compiler for no optimizations (AOT), speed optimizations (AOT-Sp), size optimizations (AOT-Si) and both speed and size optimizations (AOT-SpSi).

Much of the work presented in the TakaTuka VM (Aslam et al., 2010) focuses on bytecode compaction (using single and multiple instruction compaction, and also constant pool optimization). Java also provides textual descriptions of each class, function and field (in order to allow dynamic loading). The TakaTuka VM does not remove any class, function and field string names unlike the other proposed approaches including the work presented in this thesis. It has been decided to opt for such a technique since such textual representations can provide little benefit for on-node applications. It is most likely for this reason that the TakaTuka VM bytecode is in most cases larger than that of the bytecode produced for the AOT version presented in this thesis. That said, this is a design choice, and either of the proposed solutions could choose to keep or remove textual representations and any other optimisations. Both interpreters and run-time compilers can be made to operate on the same bytecode.

Bytecode is in general smaller than native code due to the higher abstraction level, except for when textual names of the code is included in the bytecode since native code does not contain such textual representations of units of code. The bytecode generated for the AOT compiler is smaller than that of native C code generated for most of the benchmarks, except for the case of the MD5 application. By analysing the Java bytecode generated for the MD5 application it can be noted that certain Java instructions result in larger code than that of native code generated from C, in particularly for any code related to references (objects or arrays), since in Java these references must first be resolved before getting access to the actual data member, whilst using the C version, all variables are statically linked. Therefore, a comparison with C is an unfair comparison due to the nature of statically linked variables. However, if a bytecode compaction algorithm were to be used the bytecode could be drastically reduced (Aslam et al., 2010) which would result in a much smaller size than that of the native code generated by C. That said, bytecode compaction is not the research contribution of this thesis. It must be noted that the bytecode can be the same for that of an interpreter or an AOT compiler.

5.2.2 Virtual Machine and Application Size Evaluation

An analysis of the program encoding size has been presented (i.e. the bytecode or native code for the C and VM based implementations), however this only represents the program logic and does not include the facilitating framework to support the program logic. Since sensor nodes have a limited amount of program space it is necessary to evaluate the total program space required for such applications which also includes the virtual

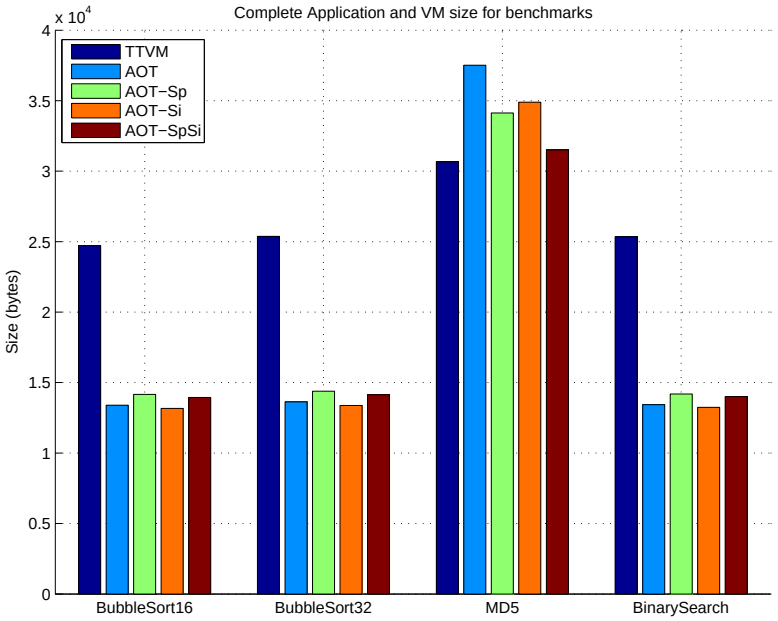

Figure 5.5: The size of the complete application and the VM footprint for TakaTuka (TTVM), and the AOT compiler for no optimizations (AOT), speed optimizations (AOT-Sp), size optimizations (AOT-Si) and both speed and size optimizations (AOT-SpSi).

machine. Virtual machines provide a number of benefits which include portability, a higher level abstraction which results in an easier programming paradigm and (usually) a smaller program encoding (which will provide benefits when sending code over the air). The C based implementations do not provide any framework to support this and thus it would be unfair to compare the C based implementation with the virtual machines. In Figure 5.5 the total size of the virtual machine plus the application logic for the TakaTuka Virtual Machine (TTVM), and the AOT compiler configurations for no optimizations (AOT), speed optimizations (AOT-Sp), size optimizations (AOT-Si) and both speed and size optimizations (AOT-SpSi) are compared.

The TakaTuka virtual machine keeps the bytecode stored in the same format that the bytecode was generated (and thereafter interprets the bytecode). The compiler generates the virtual machine and tailors the VM according to the bytecode used in the application. Thus, if any bytecode commands are not used, then the interpreter will not be compiled with such logic and thus will decrease the footprint of the interpreter. However, in doing so it would be impossible to update deployed sensor nodes with code that contains commands that the interpreter was not built for. The AOT compiler, on the other hand, translates the bytecode to native code. The native code generated will be substantially larger than that of the bytecode. More so, the AOT compiler is compiled to include all possible bytecode commands so that deployed sensor nodes can be updated with new application logic without any problems. That said, either approach could be used for both interpreters and AOT compilers.

Table 5.1: System Footprint

Component	Program Size (bytes)
AOT Compiler	7414
Flash Controller	106
Garbage Collector	400
Run-Time Environment	3872
Threading	494

Looking at Figure 5.5, surprisingly, the total size of the TakaTuka virtual machine and the application logic (which is encoded as bytecode) is larger for most of the cases results when compared with the AOT compiler footprint and the native code application logic. Only when the application logic is substantially large, does the interpreter result in a smaller footprint. This is most likely due to the interpreter having a large standard footprint. Also, since the interpreter removes any code related to bytecode commands which are not used, it is expected to see a much smaller footprint for the interpreter. That said, it would be interesting to see how the results compare using the same virtual machine generation approach (i.e. either both include the full bytecode command set, or both remove any code related to unused bytecode commands). Even for a substantially larger application such as the MD5 application, the total footprint of the AOT compiler is not much larger than that of the interpreter.

A breakdown of the run-time compilation system footprint is provided in Table 5.1. The AOT compiler is the largest component with a footprint of 7500 bytes. In order to program code to flash, a flash controller is implemented which requires 112 bytes of program space. That said, the TakaTuka virtual machine does not provide a flash controller, and therefore if flash storage would be required in a TakaTuka application, a flash controller would be required to be included (and increase the TakaTuka footprint). The *naïve* mark and sweep garbage collector require 240 bytes of program space and threading requires 494 bytes. The run-time environment, which consists of functions used by the underlying system during program execution (including memory management, arithmetic and other system level functionality) requires 3020 bytes of program space.

5.3 Garbage Collection Evaluation

In aim of evaluating the whole system a garbage collection evaluation was conducted. Garbage collection is (usually) implemented in the run-time environment in a language that is compiled to native instructions. Therefore, given the same amount *garbage*, both an interpreter's and a run-time compiler's garbage collection should result in roughly the same overhead. An evaluation on how long garbage collection takes on a reasonably large amount of garbage was conducted by creating a number of objects that immediately become garbage. The test code is as follows:

Table 5.2: Garbage Collection Evaluation

Test	RTC Time (ms)	TakaTuka Time (ms)
Create 400 x 4 byte objects and GC	184	1899
Create 400 x 4 byte objects	142	1824
Garbage Collect 400 x 4 byte objects	43	75

```
public void createGarbage() {
  for(int i = 0; i < 400; i++)
    new Garbage();
}
public void runTest() {
  createGarbage();
  System.gc();
}
```

The `Garbage` class defintion consist of 2 short fields as follows:

```
public class Garbage {
  public short garbage1;
  public short garbage2;
}
```

The test creates 400 objects consisting of 4 data bytes (plus object meta-information bytes). As soon as the `new Garbage();` line is executed the object created immediately becomes garbage since it is unreachable. After creating the garbage, the garbage collector is invoked. The time taken to perform the whole process was analysed in a similar fashion to the tests presented at the beginning of this chapter (i.e. by using an LED to indicate the beginning and end of a test, and monitoring the current consumption of the system on an oscilloscope). The results are presented in Table 5.2. The run-time compiled version took 184 milliseconds to complete, whilst the TakaTuka virtual machine took 1899 ms, that is 10.32 times longer. However, the test above does not provide enough fine-grained information to calculate the overhead of garbage collection. Therefore, the time taken to create the objects only was analysed in which the TakaTuka virtual machine took 12.84 times longer. The actual garbage collection time was analysed in isolation to reveal that the mark-and-sweep implementation used in the work presented in this thesis took 43 ms to execute, whilst the mark and compact implementation used in TakaTuka took 75 ms. The difference in time is justified since the mark and compact implementations requires (over and above the mark and sweep tasks) an additional compact phase. However, the exact same garbage collection implementation could be used for both systems. Therefore, assuming that the TakaTuka virtual machine requires 43 ms to perform the same garbage collection, then the whole implementation would take 10.15 times longer (rather than 10.32 times longer). Essentially, garbage

```
//fake sample
setLedOn(0);
waitMs(1);
setLedOff(0);

//store reading
synchronized(vals) {
  vals[cnt] = cnt;
  cnt++;
}

waitMs(1000);
```

Figure 5.6: Thread code to sample a sensor and store the values. Sensor sampling is represented by LED toggling to remove any unfair comparisons due to driver implementation.

```
waitMs(10000);

//calculate average
average = 0;
synchronized(vals) {
  for(short s = 0; s < cnt; s++) {
    average += vals[s];
  }
  average = average / cnt;
  cnt = 0;
}

//fake transmission
setLedOn(1);
waitMs(1);
setLedOff(1s);
```

Figure 5.7: Thread code to average readings and transmit the result. Radio transmission is represented by LED toggling to remove any unfair comparisons due to driver implementation.

collection is a common factor for both interpretation and run-time compilation. The execution time for both will (or should) be increased by the same quantity for the same application. This thesis is not concerned with garbage collection implementation details, however the subject is definitely of interest and further analysis in respect to the ideal garbage collection implementation for WSN applications should be conducted in future work.

5.4 Threading Evaluation

Threading can be used to allow application designers to decompose an application into different tasks. Although, the threading implementation is not a primary contribution of this work but rather just a demonstration of the fact that run-time compilation can also support threading, an experimental evaluation is provided to highlight the programming benefits of threading. A comparison between a threaded application for both the AOT compilation scheme and the TakaTuka virtual machine will follow. Assuming a typical WSN application that samples a sensor, performs some computation on readings, and periodically sends out the result was used. In order to remove any unfair comparisons due to driver implementation an LED is used to represent sensor sampling and radio transmission. Two threads are used, one for sampling the sensor and storing readings as outlined in Figure 5.6, and the other for averaging the readings and transmitting the result as outlined in Figure 5.7. The synchronized blocks are used so that the common readings storage array is not altered by one thread whilst the other is using it. The experiment was conducted to determine the total energy consumed for a single cycle (i.e. 10 sample thread executions and a single send thread execution). The sensor node was connected to an oscilloscope to monitor the current consumption, and the derived energy consumption for the cycle resulted to 59 mJ and 69 mJ for the AOT compilation and interpreted versions respectively. This experiment shows that for a threaded application (which provides a higher level programming tasks abstraction) the AOT compilation approach results in 17% less energy consumption than that of the interpreted version, for an application that is most of the time sleeping. For applications with more extensive processing, obviously, the gains would be much larger.

5.5 Evaluation Discussion

In this chapter an evaluation of the execution overhead and program size for both an interpreter designed for sensor nodes (TakaTuka) and the AOT compiler has been provided. It is evident from the program execution overhead evaluation that the AOT compiler results in substantially less execution overhead which in turn implies a longer node lifetime (as discussed in Chapter 3).

Although, a substantially larger footprint is expected for AOT compilation, due to: native code being larger than bytecode; the fact that the TakaTuka virtual machine removes code to do with unused bytecode (whilst the AOT compiler does not); and also the decrease in bytecode in TakaTuka by using bytecode compaction, it turns out that the implementation size and generated native code is in some cases smaller than that of the interpreter, and only for larger applications results in a slightly larger footprint.

That being said, since the implementation results in less execution overhead, whilst at the same time requiring less or comparable program space, it has been shown that run-time compilation of bytecode for sensor networks is in fact possible, feasible, and practical contrary to the general consensus in the research field (Palmer, 2004; Koshy and Pandey, 2005; Pandey and Koshy, 2006; Koshy et al., 2008; Aslam, 2011).

Although the footprint of the AOT compiler and resultant applications is already comparable with that of interpreters, in the next chapter efforts for providing a smaller application size footprint by using Just-in-Time (JIT) compilation for resource constrained devices will be presented. JIT compilation will allow the compiler to store bytecode and only generate native code when it is required to be executed.

Chapter 6

Enabling JIT Compilation for Resource Constrained Devices

In the previous chapter the implementation of an Ahead-of-Time (AOT) compilation technique for resource constrained devices has been provided. The results show substantially better execution costs for AOT compiled code against interpreted code. More so, the footprint for the virtual machine and application code is comparable and in some cases it is even smaller than that of an interpreter. In aim of further decreasing the application footprint a Just-In-Time (JIT) compiler could be used. This chapter presents a proposed method in aim of enabling JIT compilation for resource constrained devices such as sensor networks.

Traditional platforms that perform Just-In-Time compilation typically compile whole functions at a time (Krall and Grafl, 1997; Yang et al., 1999; Brandner et al., 2009). Compiling whole functions at a time will consume a substantial amount of memory, a resource which is severely limited in such resource constrained devices. It is most likely due to this reason, that the general consensus is that JIT compilation is impossible and impractical for such resource constrained devices (Palmer, 2004; Koshy and Pandey, 2005; Pandey and Koshy, 2006; Koshy et al., 2008; Aslam, 2011). However, there is no reason why code cannot be compiled and executed in smaller granularity. In fact, Hennessy and Patterson (2006) show that statistically 90% of execution time is spent in 10% of code, and therefore it is not necessary to keep all code compiled to native code since a majority of the code will only be used rarely. Also, code is typically stored in flash memory on such microcontroller devices and executed thereafter. Flash programming does require substantial power and time, and thus constant flash programming would decrease the lifetime of such a system and increase the start-up time for the execution of JIT compiled code. This is another reason why JIT compilation is deemed unfit for such devices, as pointed out by Aslam (2011). However, contrary to the statement "JIT would require generation of native code in flash memory" (Aslam, 2011), there is

no restriction which limits native code to being generated in flash memory, and it can in fact be generated and executed from RAM (Texas Instruments, 2004). A proposed method to enable JIT compilation for such resource constrained devices will now be presented, by using Basic Block JIT Compilation, Direct JIT Compiler Calls, and a Circular JIT Cache.

6.1 Basic Block JIT Compilation

Compiling code from bytecode to native code is usually performed as functions are required and are compiled at the granularity of a function. Traditional platforms that perform JIT compilation for Java compile functions as a whole. If JIT compilation were to use flash memory as its means of storing executable code, a high power consumption ovrhead would be incurred as well as a delayed start due to flash programming time. Typical microcontrollers used in sensor networks allow for code to be executed from RAM which would not incur any extra overhead in the translation process, however RAM is scarce in such systems. Therefore, a Basic Block JIT compilation scheme is proposed, which will allow code to be JIT compiled and executed at the granularity of basic blocks.

The definition of a basic block as defined by Aho et al. (1986) is:

> A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

Basic Block JIT compilation for Java was proposed by Rogers (2002). In their thesis, Rogers (2002) shows that basic block compilation results in an 18% decrease in bytecode compilation for their evaluation test cases. When bytecode for a whole function is compiled to native code, it is possible to compile bytecode which will in fact not be used (due to code branches that are not taken due to conditional statements). By using a basic block JIT compilation scheme the memory required to store the generated native code and translation process will be minimized.

6.1.1 Offline Basic Block Analysis

Traditionally, basic block analysis is performed by the JIT compiler prior to beginning the translation process. Once the basic blocks have been identified, the compiler can then compile each basic block as the basic block is executed. In order to minimize the footprint of the JIT compiler, basic block analysis has been removed from the JIT compiler stage, and instead offline basic block analysis is performed at the source code

compilation stage when converting from Java bytecode to the intermediate bytecode (described in Chapter 4). The intermediate bytecode will then be altered to include 'start basic block', `sbb`, bytecode commands which are used to represent the start of a basic block. The 'start basic block' bytecode command will also be used to represent the end of a preceding basic block. Consider the following function which calculates the Greatest Common Divisor using the Euclidean algorithm:

```
private static short gcd(short a, short b) {
  while(a != b) {
    if (a > b) {
      a = (short)(a - b);
    }
    else {
      b = (short)(b - a);
    }
  }
  return a;
}
```

When compiled to Java bytecode this will produce the following:

```
 0    iload_0
 1    iload_1
 2    if_icmpeq 24
 5    iload_0
 6    iload_1
 7    if_icmple 17
10    iload_0
11    iload_1
12    isub
13    istore_0
14    goto 0
17    iload_1
18    iload_0
19    isub
20    istore_1
21    goto 0
24    iload_0
25    ireturn
```

As previously discussed, basic block identification is performed before JIT compilation to release the burden of basic block analysis from the resource constrained device. Therefore, when the intermediate bytecode is being generated from the above bytecode, the

`start basic block` bytecode commands will be inserted into the intermediate bytecode to produce the following:

```
 0    sbb
 1    sload_0
 2    sload_1
 3    if_scmpeq 28
 6    sbb
 7    sload_0
 8    sload_1
 9    if_scmple 20
12    sbb
13    sload_0
14    sload_1
15    ssub
16    sstore_0
17    goto 0
20    sbb
21    sload_1
22    sload_0
23    ssub
24    sstore_1
25    goto 0
28    sbb
29    sload_0
30    sreturn
```

One of the converter's job is to translate from the 32 bit width stack which is used by Java to a 16 bit stack width which is more suitable for such devices. The change in stack width is noted above from integer operations to short operations (e.g. from `iload` instructions to `sload` instructions). The converter is described more in detail in Chapter 4. Back on topic, the bytecode instruction `sbb` (in lines 0, 6, 12, 20 and 28) represents the start of a basic block. From the bytecode above it can be seen that all jump locations are resolved to a `sbb` instruction. Thereafter, when JIT compilation is required, the device is only required to compile from the beginning of the basic block to the end of the basic block, and then execute the compiled code. This mechanism will reduce the amount of memory that is required to both perform JIT compilation as well as store the generated native code. More so, very large basic blocks can be broken up into smaller blocks by inserting an `sbb` bytecode instruction where required. This could be useful if the allocated memory for JIT compilation is restricted to less than the size of a particular basic block. An alternative to this would also be to compile such large

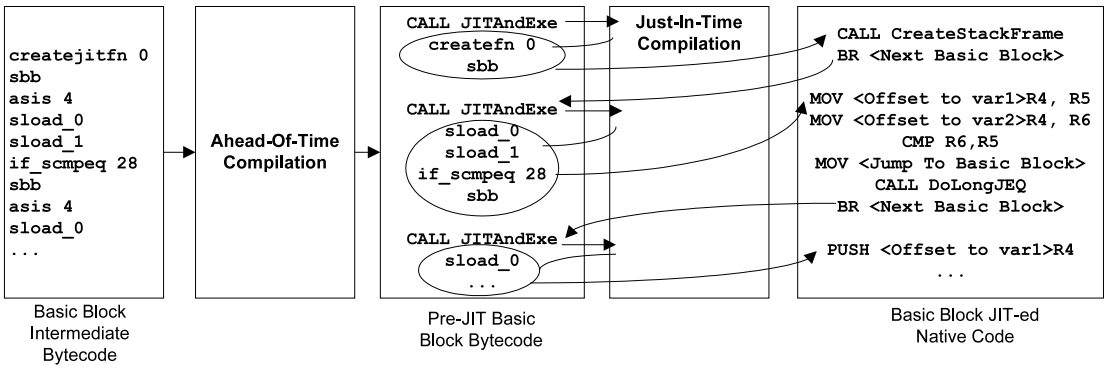

Figure 6.1: This figure demonstrates intermediate bytecode with 'start basic block' instructions compiled to basic block bytecode which is stored on the device and prepared for JIT compilation. When the JIT compiler requires to compile and execute code, it will compile only the basic block which is to be executed and then compile (and execute) additional basic blocks as they are executed.

basic blocks to flash, and thereafter keep the basic block's native code in flash as long as possible (in order to minimize flash writes due to the overhead).

6.1.2 Supporting a JIT Compiler

In Chapter 4 a method to enable run-time compilation of bytecode for sensor networks by using an Ahead-Of-Time (AOT) compiler was presented. When using the AOT compiler, bytecode is translated to native code upon receiving the bytecode (whether over a wired or wireless connection). The AOT compiler, besides compiling bytecode instructions to native code, also generates native code to support function creation and destruction (consisting of function stack frame maintenance code) as described in Chapter 4. The logic used in AOT compilation is essentially the same for JIT compilation, except for the timing at which the compilation to native code occurs. In supporting JIT compilation, changes would need to be made to the Ahead-Of-Time compiler and the bytecode transmission protocol to be able to identify which parts of bytecode are to use JIT compilation and which parts are to use AOT compilation. In doing so, mixed-mode compilation can be enabled along with an execution paradigm which can provide AOT compiled code for heavily executed code, and JIT compilation for code which is used less frequent.

Figure 6.1 depicts the compilation process from the received intermediate bytecode down to the native code which is compiled and executed by the Just-In-Time compiler. When the Ahead-Of-Time compiler receives bytecode which is intended for JIT compilation, it creates a native code execution call to the JIT compiler for each basic block. Thereafter, when the JIT compiler is called (from the native code execution calls just generated) it will compile the bytecode associated with the basic block to native code and then execute

it (this execution process is further discussed in Section 6.2). As previously mentioned the process for AOT and JIT compilation is the same, and thus the same compiler code is used for both processes. However, due to the different timings of the events, differentiation is made between the two in Figure 6.1 as Ahead-Of-Time Compilation and Just-In-Time Compilation (but are not actually different compilers).

In order to prepare code for JIT compilation at the AOT compilation stage it was required to add a new bytecode command, `createjitfn` which differentiates a function which is intended for JIT compilation. Also, as previously described, the `sbb` bytecode instruction will be inserted at the start of each basic block. A `sbb` instruction is not required at the start of a function since the basic block is implicit at the beginning of a function (however, in the example above the first instruction is also a jump destination and thus the instruction is required). When the `createjitfn` is encountered at the AOT compilation stage, the compiler will create native code to call the JIT compiler, followed by the standard AOT `createfn` instruction so that the JIT compiler can correctly create the function when it is invoked.

The `sbb` command will instruct the AOT compiler to insert the command at the current basic block to identify the end of the current basic block and the beginning of the next basic block. Thereafter, similar to the `createjitfn` instruction, the AOT compiler will create a native function call to the JIT compiler so that the next basic block can be JIT compiled and executed. Since the compiler by default will compile bytecode received to native code, it is necessary to instruct the compiler to treat bytecode intended for JIT compilation to be stored as is, so that it is not compiled Ahead-Of-Time, but compiled Just-In-Time, and therefore the bytecode instruction `asis` was added, which instructs the compiler to copy bytecode as is (up to the specified number of bytecode instructions).

6.2 Direct JIT Compiler Calls

When code which is required to be prepared for JIT compilation is loaded into the system a native function call to the JIT compiler, `JITAndExe`, is generated and placed above the related bytecode. In doing so, any code which calls such JIT enabled functions, is only required to make a call to the native call statement location which initiates the JIT process. Thus, the callee is unaware of whether the function has already been compiled to native code or if it is intended for JIT compilation and thus can enable a mixed compilation paradigm.

Figure 6.2 depicts the execution process for a function prepared for JIT compilation. The function call stack for each step of the execution process is given on the right side of the figure, and each step is labelled by a dashed line. Initially, in step 1, the callee function (whether it is a function compiled Ahead-Of-Time, Just-In-Time or if it is a native function) will be executing and have its own stack. The callee then makes a `CALL`

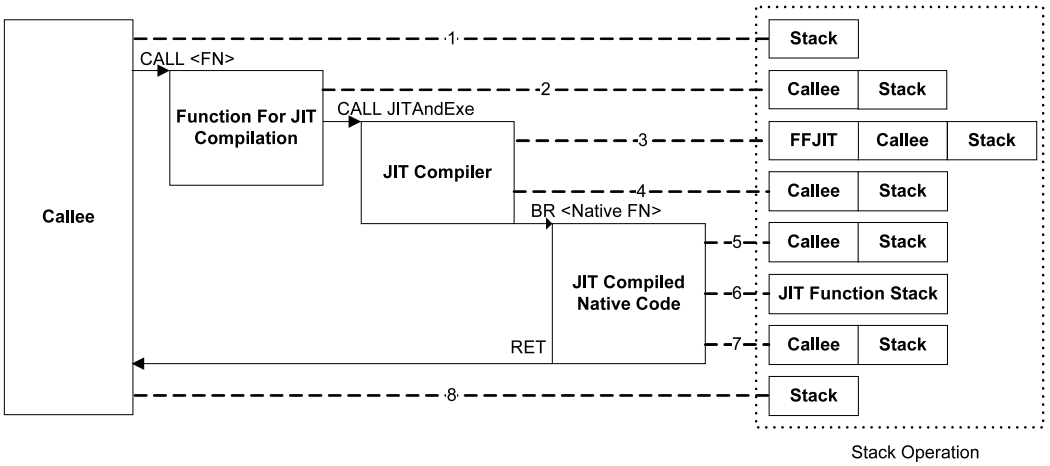

Figure 6.2: This figure shows the execution process for a function prepared for JIT compilation. The function call stack at each step of the process is presented on the right.

to the function which requires JIT compilation (however the callee does not require to know whether the function is intended for JIT compilation or if it is a native code function). The `CALL` instruction will place the callee's return address on to the stack, in step 2, so that a `RET` instruction can return to the callee later on. All functions (and basic blocks) which are intended for JIT compilation will only consist of a native code function call to the JIT compiler, `JITAndExe`, followed by the bytecode which is to be compiled to native code. Thus, the code intended for JIT compilation will then `CALL` the JIT compiler. Since the bytecode to be compiled is stored immediately after the `CALL` instruction, and the `CALL` instruction places the address to the next instruction on the stack, then in step 3, the address which is placed on the stack will be pointing to the bytecode to be compiled, i.e. FFJIT. After the JIT compiler is able to determine the address of the bytecode to compile, the address is not required on the stack. Thus, in step 4 the JIT compiler removes the address from the stack. After compiling the bytecode to native code, the JIT compiler is then required to execute the native code. Since, execution is not required to return to the JIT compiler, then there is no need to put the JIT compiler's address on to the stack, and therefore a branch, `BR`, instruction is used to change execution to the generated native code without placing any return address on the stack, as depicted in step 5. The JIT compiled function will then use it's own stack frame, i.e. the 'JIT Function Stack' depicted in Figure 6.2 for the execution of its code in step 6. Thereafter, once the JIT compiled function is done executing, as explained in Chapter 4, the function's stack frame will be de-allocated and the previous stack frame will be restored as shown in step 7. Since, the callee's address is on the stack, a `RET` instruction can be used to return to the callee. By manipulating the call stack as specified above, calls to functions intended for JIT compilation will operate in exactly the same way to other native functions. Also, there is no extra book keeping required in order to be able to facilitate JIT compilation.

6.3 Circular JIT Cache

Writing to flash consumes a substantial amount of power and time. Also, code is most frequently executed from flash. As previously mentioned this fact most likely attributed to the general consensus regarding the possibility and viability of a JIT compiler for such resource constrained devices. Although code is most typically executed from flash, contrary to the statement made by Aslam (2011), code can also be executed from RAM. RAM is, however, usually more constrained than flash and thus majority of RAM should be used for the user application.

By using a basic block JIT compilation scheme the amount of space required to execute JIT compiled code can be minimised to that of the largest basic block. For very large basic blocks native code could either be permanently or temporarily stored in flash, or could even be broken down into smaller basic blocks. Also, it is common practice to keep methods as short as possible when using an Object Oriented language such as Java (Sharp, 1996). More so, studies show that basic blocks tend to be shorter than functions (Antonioli and Pilz, 1998), although this is intuitively obvious since basic blocks cannot exceed the function size and also given the high usage of loops and conditional statements.

Previously JIT compiled code can be stored for future execution instead of having to always perform JIT compilation. A circular cache is proposed for storing JIT compiled code. A circular cache will ensure that the most recently executed JIT compiled code will be available in the cache. The reason for a "first in, first out" (FIFO) policy is due to the inherent execution path of loops which are common in code. Although, this may not be the best JIT cache policy to use, it will serve its purpose for evaluation of a JIT compiled approach which relies on a minimal amount of RAM. Further cache policies can be used and should be further investigated as a further extension of this work.

Figure 6.3 depicts an overview of the execution process for code which requires JIT compilation. A cache lookup will be performed to determine if the basic block has already been compiled and is present in the JIT cache. If it is, then the code can just be executed without incurring any extra overhead. Otherwise, the basic block native code size will be computed and thereafter the code will be compiled to native code if sufficient space is available in the JIT cache. If there is not enough space available, the oldest basic blocks in the JIT cache will be removed until there is enough space. The code can then be stored in the JIT cache (for future cache hits) and then executed. System performance will obviously be dependent upon the amount of memory reserved for the JIT cache. The more memory reserved for the JIT cache, the more likely it is that compiled code will remain in the JIT cache for future use. The current implementation requires a statically sized JIT cache. However, future work can be implemented to share memory between the application and the JIT cache. An analysis of performance based on the JIT cache size is provided in Chapter 7.

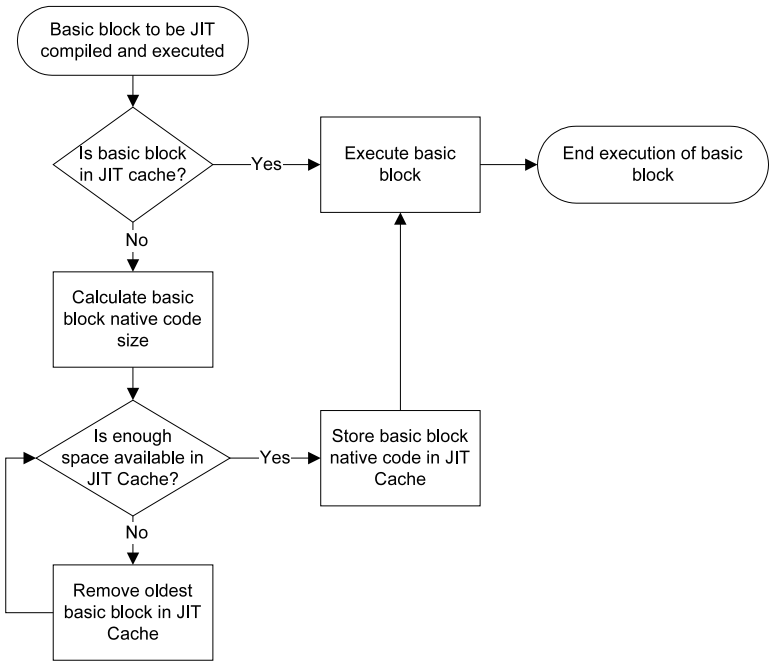

Figure 6.3: An overview of the execution process for code which requires JIT compilation.

Besides the compilation overheads inherent in JIT compilation, two other JIT cache related operations add to the overhead of JIT compilation and execution being: JIT cache lookup and JIT cache code addition. The JIT cache keeps track of code blocks by means of a lookup table and an associated lookup entry for each code block. Lookup entries consist of the location of the generated native code and the location of the source bytecode block used to generate the native code. Thereby, whenever a lookup is being performed to check whether the block being executed is already in the JIT cache, the lookup table is traversed. The worst case lookup therefore is, in Big O notation, $O(n)$ where n is the number of code blocks stored in the cache.

In Chapter 5 it has been shown that AOT compilation performs substantially better than an interpreter. In aim of decreasing resource footprint, in this chapter techniques to enable JIT compilation for severely resource constrained devices have been presented including basic block JIT compilation, offline basic block analysis, direct JIT compiler native calls and a circular JIT cache. An evaluation of the JIT compiler will now be given to deduce whether footprint can in fact be decreased.

Chapter 7

Evaluation of the JIT Compiler

Methods to enable JIT compilation for resource constrained devices were proposed in Chapter 6. An evaluation will now be provided in respect to how the JIT compilation method compares to the TakaTuka interpreter. The TakaTuka interpreter requires 862 bytes of RAM to operate for an empty application. We have therefore decided to base the JIT cache size upon this value. The JIT implementation requires 340 bytes of RAM to operate, and therefore we have decided to use a 500 byte JIT cache. Following an evaluation is provided that analyses the JIT compiler's execution speed and the size of both the bytecode encoding and bytecode storage requirements.

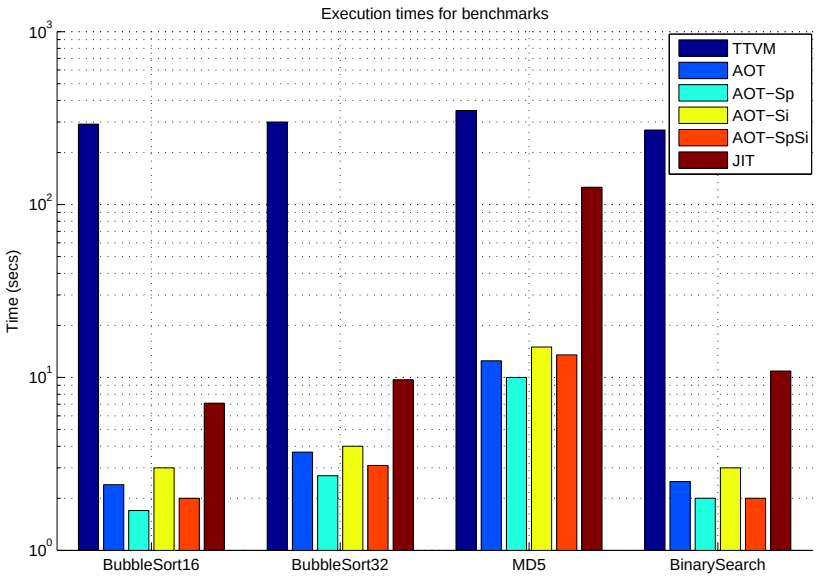

Figure 7.1: Execution times for AOT, JIT and TakaTuka versions of the benchmarks. The y axis uses a log scale to better visualise the large differences.

7.1 Execution Performance Evaluation

Figure 7.1 depicts the execution times for the TakaTuka VM (TTVM), AOT without optimisations (AOT), with speed optimisations (AOT-Sp), size optimisations (AOT-Si), both optimisations (AOT-SpSi) and JIT compilation. The JIT compiled code's worst case execution performance for the MD5 benchmark is 10 times the execution speed of the AOT compiled code, however is still half the speed of the TakaTuka VM. The other JIT benchmarks range from 2.5 to 4.5 times the AOT execution speed. The motivation behind the JIT compiler though is not performance, but is to decrease the storage overhead inherent in the AOT compilation scheme. Therefore, an evaluation on code size follows.

7.2 Program Size Evaluation

Program space on WSN nodes is limited. The evaluation of the AOT compiler program sizes in Chapter 5 demonstrate that the program sizes of the AOT compiled code is comparable with that of the TakaTuka VM. However, due to the bytecode encoding used in the TakaTuka VM it should have a substantially smaller size (although it does not remove textual class and function name representations). In any case, the native code generated for the AOT compiler is larger than that of the intermediate bytecode produced in the pre-processing stage. In aim of decreasing space requirements, JIT compilation was proposed as described in the previous chapter. Analysis on the intermediate bytecode prior or after loading into the system occurs is first provided as a means to evaluate the overhead of transmitting bytecode to sensor nodes and the individual application space overhead, followed by an analysis on the total footprint of the VM and the installed application.

7.2.1 Program Encoding Size Evaluation

Figure 7.2 provides the sizes of the TakaTuka VM (TTVM) and the intermediate bytecode (AOT-BC) along with the native code generated from the AOT compiler for no optimisations (AOT), speed optimisations (AOT-Sp), size optimisations (AOT-Si) and both optimisations (AOT-SpSi) and the altered bytecode for JIT compilation before (JITBC-BEFORE) and after (JITBC-AFTER) the bytecode is loaded into the system. As can be seen from the graph, minimal increased space is required to convert bytecode to enable JIT (by adding start basic block bytecode instructions), and a slight increase is thereafter introduced when the bytecode is stored on the node (due to direct JIT compiler native calls). The JIT enabled bytecode both before and after being loaded into the system is less than that of the interpreter except for the case of the MD5 benchmark whereby the JIT enabled bytecode after installation is slightly larger (due to direct JIT

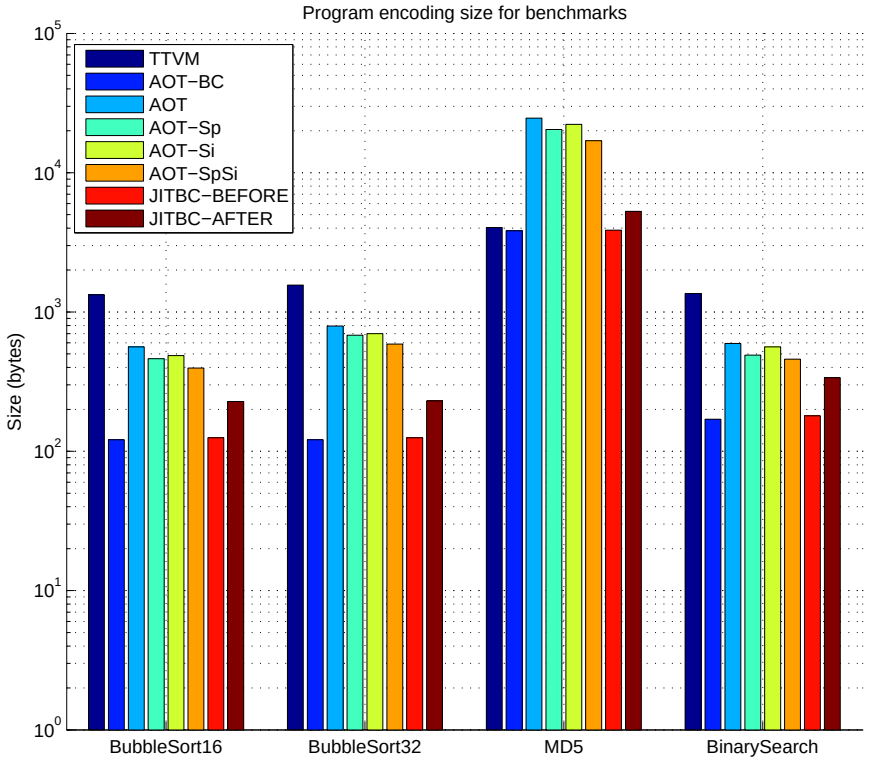

Figure 7.2: The size of the benchmark application logic for TakaTuka bytecode (TTVM), the intermediate bytecode (AOT-BC), the generated AOT compiler native code size for no optimizations (AOT), speed optimizations (AOT-Sp), size optimizations (AOT-Si) and both speed and size optimizations (AOT-SpSi), and the bytecode for the JIT compiler before (JITBC-BEFORE) and after (JITBC-AFTER) the bytecode is loaded into the system.

compiler native calls). However as explained in Chapter 5, the TakaTuka VM could just as easily produce and operate on a smaller bytecode size. The important thing to note here is that there is a slight increase in JIT enabled bytecode size prior to loading, an average of 15%, and a slightly larger increase in size after loading, an average of 33%. However, when compared to the generated native code the storage space required for that of the JIT code after loading is on average 50% of the best case AOT compilation scheme.

7.2.2 Virtual Machine and Application Size Evaluation

To put the size gains into perspective a whole system size analysis including the application storage requirement and the VM footprint will now be provided. Extra code is required to extend the AOT compiler to support the JIT compilation scheme described in the previous chapter. The added compiler footprint amounts to 1544 bytes, totalling the JIT compiler to 9044 bytes. Figure 7.3 depicts the full system size for the TakaTuka

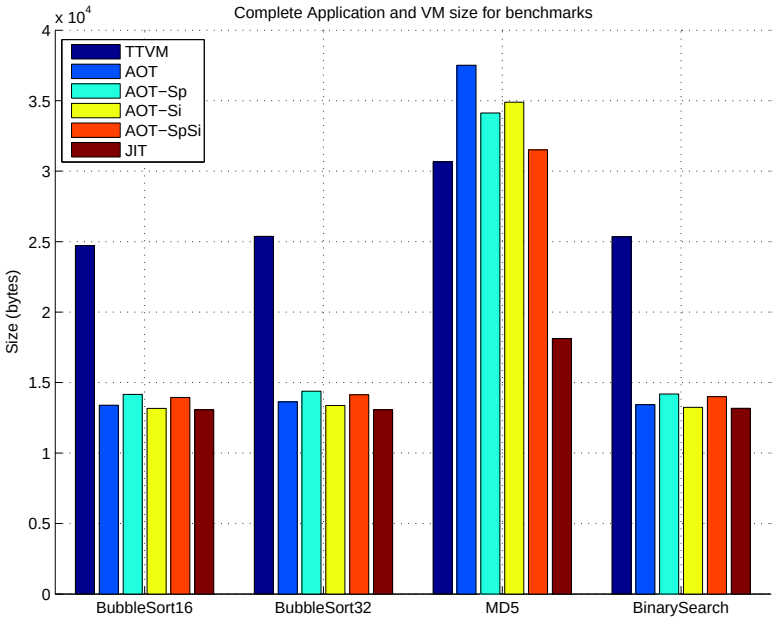

Figure 7.3: The size of the complete application and the VM footprint for TakaTuka (TTVM), and the AOT compiler for no optimizations (AOT), speed optimizations (AOT-Sp), size optimizations (AOT-Si) and both speed and size optimizations (AOT-SpSi) and the JIT compiler.

bytecode and VM footprint (TTVM), the AOT compiler footprint and generated native code for no optimisations (AOT), speed optimisations (AOT-Sp), size optimisations (AOT-Si), both optimisations (AOT-SpSi) and for the JIT enabled bytecode after loading along with the JIT compiler footprint (JIT). From the graph it can be seen that the footprint from the whole application is substantially less than that of the interpreter. Minor savings are made for the smaller benchmark applications, however large program space savings can be seen for the MD5 benchmark. In Chapter 5, the MD5 application was the only benchmark in which the interpreter required less program space. By using a JIT compilation scheme the program space overhead even for a large application such as the MD5 application results in less program space than that of an interpreter.

7.3 Benefits of JIT Compilation

Native code requires more space than bytecode. When code is translated from bytecode to native code using the AOT compiler, the amount of space required to store the generated native code greatly outweighs the bytecode encoding. For large applications there may not be enough space to store the generated native code (due to the size expansion). Java is particularly bad at expressing array data initialisation. Take for example the following array initialisation:

```
arr = new byte[] {
  127, 127, 120
```

Table 7.1: 16 bit Bubble Sort JIT Cache Analysis

JIT Cache Size	Hits	Misses
500	66300	10
300	65281	1029
250	256	66054

```
    };
```

The generated bytecode compiled using the standard Java compiler is as follows:

```
 0 iconst_3
 1 newarray 8 (byte)
 3 dup
 4 iconst_0
 5 bipush 127
 7 bastore
 8 dup
 9 iconst_1
10 bipush 127
12 bastore
13 dup
14 iconst_2
15 bipush 120
17 bastore
18 putstatic #2 <Main.arr>
21 return
```

Note, that each array element has been expanded to 4 bytecode instructions which total to 5 bytes. These bytecode instructions will further be expanded to a minimum of 20 bytes using the size optimised compilation scheme. A 256 initialised byte array will therefore require 5120 bytes of program space to store the initialisation code. An FFT algorithm was implemented using a lookup table consisting of more than 1024 bytes. When this code gets compiled to bytecode and thereafter generated to native code this results in a total of 20480 bytes. In addition to this the added logic required more than another 10 KB of generated native code and therefore the application could not be loaded onto the system using AOT. This is where JIT compilation provides a substantial benefit over AOT. The bytecode can be stored in its original form and then only compiled to native code when it is required. More so, initialisation data is only required to be executed once. The FFT application successfully fit on the JIT compiled system without a problem. In total 11,084 bytes were required to store the application (in a bytecode encoding).

Table 7.2: 32 bit Bubble Sort JIT Cache Analysis

JIT Cache Size	Hits	Misses
500	66300	10
450	65281	1029
400	256	66054

Table 7.3: MD5 JIT Cache Analysis

JIT Cache Size	Hits	Misses
3000	384245	6424
2000	213561	177108
500	117232	273437

Table 7.4: Binary JIT Cache Analysis

JIT Cache Size	Hits	Misses
500	112992	14
250	104000	9006
200	102000	11006

Table 7.5: FFT JIT Cache Analysis

JIT Cache Size	Hits	Misses
2000	4112	33
675	2954	1191
600	519	3626

7.4 JIT Cache Analysis

The performance gains of JIT compilation is dependent upon being able to reuse code previously compiled. The larger the JIT cache, the more likely code is to be found. However, the larger the JIT cache, the less memory is made available to the application. An analysis was conducted for each of the benchmark applications to evaluate the optimal JIT cache size, and how hit/miss rate is affected by decreasing the size. The results for the 16 bit bubble sort, 32 bit bubble sort, MD5, binary search and FFT benchmarks are presented in Tables 7.1, 7.2, 7.3, 7.4 and 7.5 respectively.

A JIT cache size of 500 bytes was sufficient for the optimal compilation and execution of the two bubble sorts and binary search benchmarks. The optimal speeds achieved were 2.9, 2.6 and 4.35 times slower than that of the AOT (with no optimisations) compiled code. The slowdown is due to the fact that compilation is actually occurring at execution time, whilst AOT compilation compiles code before execution upon loading. A large JIT cache size of 3000 bytes was required for the MD5 application which resulted in a 10.08 times slower speed. The FFT algorithm could not be tested against AOT since it could not be loaded on the AOT enabled system.

When the JIT cache is less than a threshold whereby it can achieve a substantial hit rate, obviously, the performance is highly degraded due to extensive recompilation. The current JIT cache scheme is implemented as a FIFO JIT cache, whereby the oldest code in the JIT cache is evicted. However, other JIT cache eviction algorithms should be investigated in the future to determine the ideal one for the different cases. Also, whether a JIT cache size is acceptable for a particular application is really dependent on the application. If an application does not use more than a few hundred bytes, larger JIT caches could be used. However, if the application requires a large amount of memory, then less memory should be reserved for the JIT cache. That said, the experimental evaluation here has shown that for typical light-weight WSN algorithms including sorting, hashing and even a complicated algorithm such as the FFT, a JIT based system which requires a maximum of 3,000 bytes to a minimum of 500 bytes is required for optimal execution. If the run-time system required 1,000 bytes and a 3,000 byte JIT cache was used, then 6,000 bytes would still be available for use by the application (which is often enough for WSN applications). However, again this is very application dependent.

7.5 Discussion

In Chapter 5 an evaluation of the AOT compilation methods proposed was provided. Results heeded show a substantial increase in execution performance when compared with an interpreter. Resources required for AOT compilation was shown to be comparable or better than that of the interpreter. However, it is believed that this may be due to implementation details of the interpreter. In fact application bytecode sizes for the interpreter should have resulted in less of a footprint, although the actual interpreter size should be comparable to the AOT compiler. Since it is believed that interpreters can achieve a smaller footprint, further work was performed into minimising resources required for run-time compilation by proposing JIT compilation. Due to the resource constraints inherent in WSN class devices, techniques to enable JIT compilation were proposed in Chapter 6 including basic block offline analysis and direct JIT compiler native calls.

In this chapter we have evaluated the JIT compilation methods proposed. Results show that JIT execution performance is degraded by 2.5 to 10 times the AOT compiled code. Lower execution speeds are sacrificed for less program space requirements. It is shown that the JIT proposed methods result in less program space requirements than the interpreter. Therefore, it has been shown that simple run-time compilation techniques can be implemented to provide an efficient execution platform with comparable or less resources than that of the available interpreter.

Chapter 8

Reprogramming Sensor Networks with Run-time Compilation

Sensor networks, like other traditional computing platforms, require updates from time to time due to bug fixes, new features, or a complete re-tasking of the devices. Virtual machines provide an easier to program abstraction and also a platform independent program encoding. Steinfeld and Carro (2009) demonstrate the benefits of using a byte-code encoding for program updates. However, as presented in Chapter 3, the overheads inherent in an interpreter greatly impact the overall lifetime (even for very long sleeping periods).

In Chapters 4 and 6 techniques to enable Ahead-Of-Time and Just-In-Time compilation for resource constrained devices were presented. Results were then presented in Chapters 5 and 7 which demonstrate the performance gains and the fact that the program space that is required is comparable and sometimes less than that of the interpreters. More so, it has been shown that the RAM required for both AOT and JIT compilation is comparable or less than that of the interpreter. To further demonstrate the gains of using a run-time compilation technique for sensor networks and also due to the necessity for reprogramming sensor nodes, a model to analyse the effects of run-time compilation for reprogramming sensor networks will now be provided.

8.1 Modelling Reprogramming Overhead

The energy required to reprogram a unit of code, E_{repro}, can be modelled as:

$$E_{repro} = E_{rx} + E_{store} \tag{8.1}$$

where E_{rx} is the energy required to receive the code and E_{store} is the energy required to store the update. Similar to Dunkels et al. (2006a), a simplified model for the energy required to receive code is used, whereas E_{rx} is proportional to the size of the code as:

$$E_{rx} = E_{rxbyte} \cdot S_{rx} \tag{8.2}$$

where E_{rxbyte} is the energy consumed for receiving a single byte over the air and S_{rx} is the number of bytes of code received. A similar equation is used to model the energy required to store the update, E_{store}, as follows:

$$E_{store} = E_{storebyte} \cdot S_{store} \tag{8.3}$$

where $E_{storebyte}$ is the energy required to store a single byte to flash memory and S_{store} is the size of the update to store to flash.

Now, equation 8.1 will be adapted to represent the reprogramming overhead for interpreted code, Ahead-Of-Time compiled code and Just-In-Time compiled code as $E_{reproint}$, $E_{reproaot}$ and $E_{reprojit}$ respectively.

Although, the bytecode generated from the TakaTuka virtual machine is larger than the bytecode generated for the run-time compilation techniques, as explained in Chapters 5 and 7 the larger bytecode is due maintaining meta-information to do with classes, functions and fields. That said, interpretation and run-time compilation could both operate on the same bytecode. Therefore, it will be assumed that the interpreter's bytecode is of the same size as that used for run-time compilation (less the bytecode overhead inherent in facilitating run-time compilation).

It will be assumed that the interpreter and JIT systems incur minimal processing overheads, since they just store the received bytecode to flash with minor modifications. Also, the interpreter stores the bytecode as it is received thus the number of bytes to be stored, S_{store} will equal that of the number of bytes received, S_{rx}. Thus, the energy consumed for reprogramming a unit of code using an interpreter can defined as:

$$E_{reproint} = E_{rxbyte} \cdot S_{rx} + E_{storebyte} \cdot S_{rx} \tag{8.4}$$

and by factoring equation 8.4:

$$E_{reproint} = S_{rx}(E_{rxbyte} + E_{storebyte}) \tag{8.5}$$

In order to support JIT compilation, basic block JIT compilation was proposed which uses an offline compilation technique to determine the basic blocks prior to transmitting

the bytecode. The resultant bytecode size including the appended basic block information was analysed and compared against the original bytecode size. The average overhead introduced by adding the basic block information for the test cases is 15% and therefore the number of received bytes, S_{rx}, must be increased by 15%.

For each `start basic block` instruction, native code which calls the JIT compiler is appended to the stored bytecode. The resultant average additional storage overhead for supporting native calls to the JIT compiler is 33% for the test cases. Therefore the storage component overhead must be increased by 33%. Thus, the energy consumed for reprogramming a unit of code using the Just-In-Time compiler can be defined as:

$$E_{reprojit} = E_{rxbyte} \cdot S_{rx} \cdot 1.15 + E_{storebyte} \cdot S_{rx} \cdot 1.15 \cdot 1.33 \tag{8.6}$$

and by factoring equation 8.6:

$$E_{reprojit} = S_{rx} \cdot 1.15(E_{rxbyte} + E_{storebyte} \cdot 1.33) \tag{8.7}$$

In order to support Ahead-Of-Time compilation, gradual compilation was proposed which gradually compiles code as it is received. In order to do this the list of jump destinations must be sent prior to a function's bytecode. The overhead for appending this information has been analysed and compared to the actual bytecode size. The average overhead for the test cases is 6.6%. Thus, the number of bytes received, S_{rx}, requires an additional 6.6%.

The Ahead-Of-Time compiler, upon receiving bytecode will instantly compile the bytecode to native code. Thus, the process of reprogramming the code update will also incur a processing component, $E_{process}$, defined as:

$$E_{process} = E_{processbyte} \cdot S_{rx} \tag{8.8}$$

where $E_{processbyte}$ is the energy required to process a single byte. Thus, the energy required for reprogramming using AOT compilation can be defined as:

$$E_{reproaot} = E_{rx} + E_{process} + E_{store} \tag{8.9}$$

The native code generated for the Ahead-Of-Time compiler using both speed and size optimizations is 3.8 times larger the bytecode size on average for the given test cases. Therefore, the number of bytes required to be stored to flash, S_{store}, will be 3.8 times that of the amount of bytes received over the air. Thus, the energy consumed for reprogramming a unit of code using the Ahead-Of-Time compiler can be defined as:

$$E_{reproaot} = E_{rxbyte} \cdot S_{rx} \cdot 1.066 + E_{processbyte} \cdot S_{rx} \cdot 1.066 + E_{storebyte} \cdot S_{rx} \cdot 1.066 \cdot 3.8 \tag{8.10}$$

and by factoring equation 8.10:

$$E_{reproaot} = S_{rx} \cdot 1.066(E_{rxbyte} + E_{processbyte} + E_{storebyte} \cdot 3.8) \tag{8.11}$$

In order to calculate the processing overhead in compiling from bytecode to native code the average time for compiling 100,000 bytes without actually writing to flash was calculated. The average processing takes 70 μs per byte. The current draw for the active microcontroller state for the Telos B node is 1.8 mA (Polastre et al., 2005). Thus, the energy consumption per byte, $E_{processbyte}$, can be worked out to 0.34 μJ (assuming a 2.7 supply voltage). Therefore, this overhead can be inserted into equation 8.11:

$$E_{reproaot} = S_{rx} \cdot 1.066(E_{rxbyte} + 0.34\mu J + E_{storebyte} \cdot 3.8) \tag{8.12}$$

In their paper, Dunkels et al. (2006a) provide the energy required to receive a single byte, E_{rxbyte}, as 21 μJ. The power consumption for storing a single byte to flash, E_{store}, can be worked out from Texas Instruments (2009) to 0.55 μJ (using a 257 kHz Flash time generator, a 3 mA supply current during programming and a 2.7 program and erase supply voltage). Therefore, these values are plugged into equations 8.5, 8.7 and 8.12:

$$E_{reproint} = S_{rx}(21\mu J + 0.55\mu J) = S_{rx} \cdot 21.55\mu J \tag{8.13}$$

$$E_{reprojit} = S_{rx} \cdot 1.15(20\mu J + 0.55\mu J \cdot 1.33) = S_{rx} \cdot 23.84\mu J \tag{8.14}$$

$$E_{reproaot} = S_{rx} \cdot 1.066(21\mu J + 0.34\mu J + 0.55\mu J \cdot 3.8) = S_{rx} \cdot 24.98\mu J \tag{8.15}$$

Equations 8.12, 8.13 and 8.14 result in a linear increase in energy consumption for reprogramming as the number of bytes to be sent increases. However, the energy cost of reprogramming cannot be looked at in isolation since the reprogramming component does not take into account the overheads inherent in each different execution paradigm. Therefore, the energy consumption of reprogramming alongside the execution overheads will now be investigated.

8.2 Modelling the Energy Consumption Lifecycle of Reprogrammed Code

The energy consumption for the given life cycle of reprogrammed code, $E_{lifecycle}$, is defined as:

$$E_{lifecycle} = E_{repro} + E_{execution} \tag{8.16}$$

where equations 8.13, 8.14 and 8.12 can be used as the reprogramming overhead, E_{repro}, and $E_{execution}$ is the energy consumed for executing the update throughout the lifecycle.

Although, the energy consumed for executing an update requires an analysis of the actual code (due to conditional and loop instructions), for simplicity's sake it will be assumed that the size of the update is directly proportional to the execution cost of the update. This assumption will suffice, since the execution path taken by the interpreter, Ahead-Of-Time compiler and Just-In-Time compiler will be the same. Thus, the energy consumed for executing the update throughout the lifecycle, $E_{execution}$ can be defined as:

$$E_{execution} = \sum_{i=0}^{n-1} S_{rx} \cdot E_{execbyte} \tag{8.17}$$

where $E_{execbyte}$ is the energy consumed for executing a byte (i.e. a single byte pertaining to the bytecode which was sent to the device) and n is the number of times the code is executed. In order to determine $E_{execbyte}$, the execution overhead per byte, the execution time has been averaged for the BubbleSort16 test case by performing loop unrolling, which works out to 27 μs per byte. Thus, by using the time required to execute a byte, the current consumption and the voltage for the device (a current draw of 1.8 mA (Polastre et al., 2005) and assuming a 2.7 supply voltage) it can deduced that the energy consumed for executing a byte, $E_{execbyte}$, is 0.13 μJ. Thus, the execution cost for the Ahead-Of-Time compiler, $E_{executionaot}$, can be defined as:

$$E_{executionaot} = \sum_{i=0}^{n-1} S_{rx} \cdot 0.13\mu J \tag{8.18}$$

From the evaluation conducted in Chapter 5 it was shown that the TakaTuka interpreter is on average 94 times slower than that of the Ahead-Of-Time compiler using both speed and size optimizations for the given test cases. The Just-In-Time compilation method was further evaluated in Chapter 7 in which it was shown that the Just-In-Time compiler's worst case resulted in execution decrease of 10 times than that of the AOT compiler (with both optimizations set). Therefore, the execution costs for the interpreter and just-in-time compiler will be multiplied by the execution speed overhead. Thus, the execution cost of the interpreter and JIT compiler, $E_{executionint}$ and $E_{executionjit}$ respectively, can be defined as:

$$E_{executionint} = \sum_{i=0}^{n-1} S_{rx} \cdot 0.13\mu J \cdot 94 = \sum_{i=0}^{n-1} S_{rx} \cdot 12.22\mu J \tag{8.19}$$

$$E_{executionjit} = \sum_{i=0}^{n-1} S_{rx} \cdot 0.13\mu J \cdot 10 = \sum_{i=0}^{n-1} S_{rx} \cdot 1.3\mu J \tag{8.20}$$

Therefore, the total energy consumption for the lifecycle of reprogrammed code for an interpreter, AOT compiler and JIT compiler is defined as $E_{lifecycleint}$, $E_{lifecycleaot}$ and

$E_{lifecyclejit}$ respectively as follows:

$$E_{lifecycleint} = S_{rx} \cdot 21.55\mu J + \sum_{i=0}^{n-1} S_{rx} \cdot 12.22\mu J = S_{rx}(21.55\mu J + \sum_{i=0}^{n-1} 12.22\mu J) \quad (8.21)$$

$$E_{lifecycleaot} = S_{rx} \cdot 24.98\mu J + \sum_{i=0}^{n-1} S_{rx} \cdot 0.13\mu J = S_{rx}(24.98\mu J + \sum_{i=0}^{n-1} 0.13\mu J) \quad (8.22)$$

$$E_{lifecyclejit} = S_{rx} \cdot 23.84\mu J + \sum_{i=0}^{n-1} S_{rx} \cdot 1.3\mu J = S_{rx}(23.84\mu J + \sum_{i=0}^{n-1} 1.3\mu J) \quad (8.23)$$

Since equations 8.21, 8.22 and 8.23 are all proportional to the number of bytes in the original bytecode, then in order to compare the three, S_{rx} can be removed from the plots. Therefore, a comparison of the equations for reprogramming a single byte of bytecode for an interpreter, Ahead-Of-Time compiler and Just-In-Time compiler, $E_{reprobyteint}$, $E_{reprobyteaot}$ and $E_{reprobytejit}$ respectively are:

$$E_{reprobyteint} = 21.55\mu J + \sum_{i=0}^{n-1} 12.22\mu J \quad (8.24)$$

$$E_{reprobyteaot} = 24.98\mu J + \sum_{i=0}^{n-1} 0.13\mu J \quad (8.25)$$

$$E_{reprobytejit} = 23.84\mu J + \sum_{i=0}^{n-1} 1.3\mu J \quad (8.26)$$

Equations 8.24, 8.25 and 8.26 result in a linear relationship with the number of execution cycles, and even after a single execution cycle the dominant factor will be the execution overhead. Wireless sensor networks, however do not just perform computation, but also are required to sense the external environment and send wireless messages. Therefore, in the next section an extension to the reprogramming model to include an element of sensing and wireless communication will be provided.

8.3 Including Sensing and Wireless Communication Overheads

Equation 8.17 expresses the execution overhead component for the lifecycle of a given unit of reprogrammed code, $E_{execution}$, as consisting of the summation of execution cycles. Where each cycle is the product of the size of the bytecode and the overhead required to execute each byte. In order to add the sensing overhead, E_{sense}, and wireless

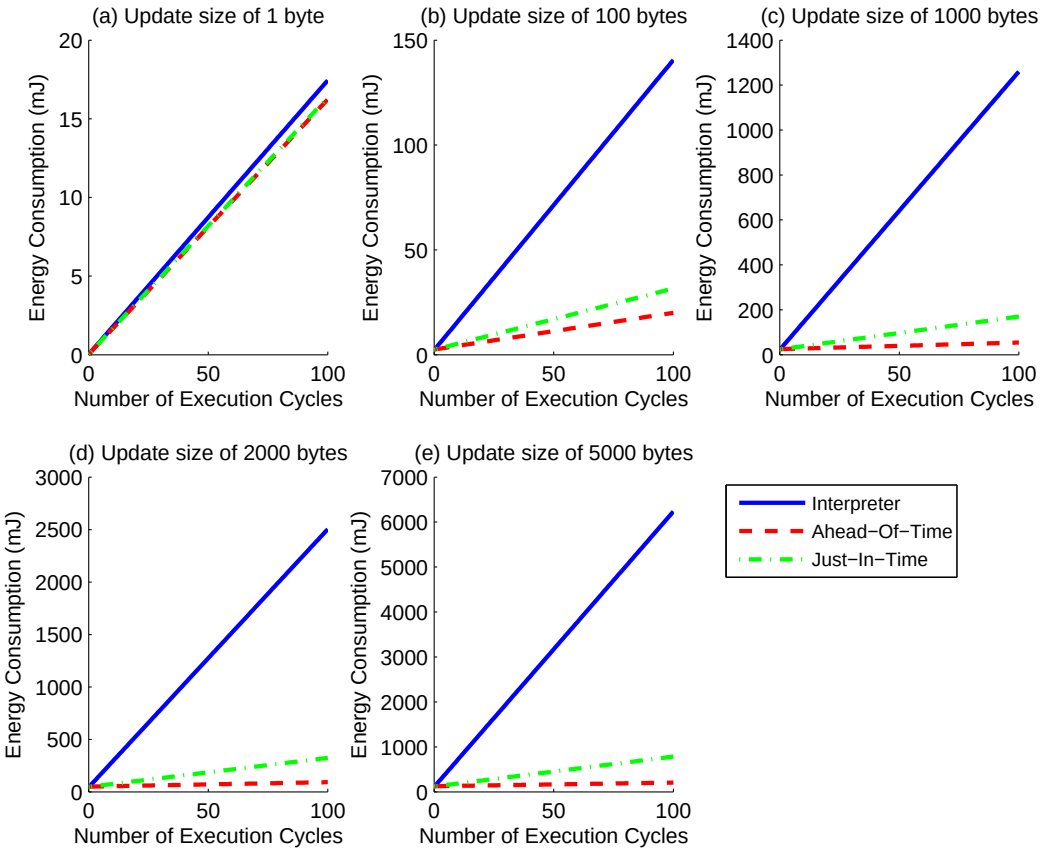

Figure 8.1: A sensing and wireless transmission consumption component is factored into the lifecycle of a reprogrammed unit of code. This figure shows the energy consumed for reprogramming and executing a unit of code for a varying number of execution cycles for (a) 1 byte, (b) 100, (c) 1000, (d) 2000 and (e) 5000 bytes.

communication overhead, $E_{wireless}$, the equation is extended as follows:

$$E_{execution} = \sum_{i=0}^{n-1} (S_{rx} \cdot E_{execbyte} + E_{sense} + E_{wireless}) \tag{8.27}$$

The actual overhead required to sense and transmit (and receive) wireless is entirely application dependent. Also, after applying the update the actual processing will most likely include previous logic already installed on the node. However to simplify the problem, let's assume that the new code being sent is the only logic to be executed. Let us assume that each time the cycle is executed a single sensor is sampled and the associated data is sent over the air. This will suffice to demonstrate the impact of sensing and wireless transmission to the model. Therefore, let's assume energy consumption for the sensor and radio as 80 μJ (Sensirion, 2010) and 82 μJ (Texas Instruments, 2010), having a combined consumption of 162 μJ. Thus, the energy consumed for the lifecycle of the unit of reprogrammed code for the interpreter, AOT and JIT compilers would

need to be rewritten as:

$$E_{lifecycleint} = S_{rx} \cdot 21.55\mu J + \sum_{i=0}^{n-1} (S_{rx} \cdot 12.22\mu J + 162\mu J) \tag{8.28}$$

$$E_{lifecycleaot} = S_{rx} \cdot 24.98\mu J + \sum_{i=0}^{n-1} (S_{rx} \cdot 0.13\mu J + 162\mu J) \tag{8.29}$$

$$E_{lifecyclejit} = S_{rx} \cdot 23.84\mu J + \sum_{i=0}^{n-1} (S_{rx} \cdot 1.3\mu J + 162\mu J) \tag{8.30}$$

Figure 8.1 depicts the curves for equations 8.28, 8.29 and 8.30 for update sizes, S_{rx}, of (a) 1 byte, (b) 100, (c) 1000, (d) 2000 and (e) 5000 bytes. When the update size consists of only 1 byte the energy consumed for the duty cycle is saturated by the sensing and wireless component. However, it is assumed that it is only the newly sent code which will be executed it is impossible to have a program size of 1 byte which consists of sampling a sensor and sending the data (unless a very high abstraction layer is used in which the single byte represents this logic). When using a Java bytecode paradigm, typical applications would require much more logic than a single byte. As the number of bytes in the update increases to 100 bytes (which is a more probably program size) the large difference in energy consumption between the execution paradigms is obvious.

8.4 An Experimental Evaluation of Reprogramming with Run-time Compilation Techniques

An experimental evaluation to determine the true costs of reprogramming with run-time compilation techniques was conducted. To determine the individual radio, processing and flash storage components involved in reprogramming a blink application was reprogrammed over-the-air three times. The first time the node would only receive the data. The second time the node would receive the data and process the bytecode instructions. The final time the node would receive the data, process the bytecode instructions and store the required data to flash. When using a radio to transmit data, different runs can result in a different outcome due to a number of factors including radio propagation. Therefore, the blink application is ideal since it is encoded in 45 bytes which is sent in a single radio message, and therefore once the first packet is received the whole reprogramming process can take place. The node was connected to an oscilloscope to monitor the current consumption, and as per other experiments in this thesis, a LED was used to indicate the start and end of the experiment. The radio current consumption for each byte could then be analysed using the first test which only receives the bytecode instructions over the radio. The radio energy consumed per byte was analysed to be 27.2μJ (which is similar to the value used in the model above). The next test involved receiving

and processing the received bytecode instructions. Using the difference between the two tests, the processing energy consumed was analysed to be 0.24μJ per byte (which is close to the value used in the model above). The final test which included receiving, processing and storing the processed instructions was conducted. The difference between the last two tests was used to calculate the flash reprogramming energy per byte to 3.4μJ (which is higher than the value used in the model above). The values deduced are not completely accurate since the difference between the tests will not exactly define the different components. However, the values are close enough to the expected values, and therefore in the very least it has been shown that for a single radio reprogramming message the model above will hold.

The case will be very different for applications having more than a single radio message though. The reason for this is due to the underlying MAC protocol used for reprogramming (which will most likely include acknowledgement messages sent from the node amongst other MAC related components). Therefore, calculating the individual reprogramming components will not be possible since the measurements will be diluted with varying energy consumption due to the changing properties of the wireless medium. However, the test can be repeated a number of times to establish an average consumption for the whole reprogramming overhead for the AOT and JIT run-time compilers. A WSN application to sample, average readings and send data will be used for this experiment. The code is outlined in figure 8.2.

The algorithm models sampling and sending in the form of a toggling of an LED. The reason why this was used is due to a fair comparison with the TakaTuka virtual machine, since as shown in Chapter 3, driver implementation can account for a large different in performance. The application will wait 1 second between samples, and every 60 samples calculate the average sample and send it out over the air. Also, since sensors are not being sampled (for a fair comparison) the sensor values must be generated. Therefore, the sample number will be used. In any case, the processing overhead will be the same for real or generated values. The resultant bytecode is 181 bytes for AOT compilation and 278 bytes for JIT compilation. Reprogramming was attempted 10 times for each AOT and JIT compilers, and the average whole programming cost per byte resulted to 1.12 mJ and 1.09 mJ for AOT and JIT respectively. The reprogramming overheads per byte are larger than that of the previous experiment since multiple radio messages are required as well as a reprogramming MAC protocol. Nonetheless, for a given MAC protocol the cost per radio message should be roughly equal. Unfortunately, the TakaTuka virtual machine does not support reprogramming so we'll assume a similar energy consumption of 1.09 mJ per byte for an interpreted version. The TakaTuka VM also entails a much larger bytecode encoding, and therefore we'll assume a bytecode encoding similar to that of the AOT of the amount of 181 bytes. The reprogramming energy consumed for AOT, JIT and the interpreter is therefore 203 mJ, 303 mJ and 197 mJ respectively. The complete reprogramming and execution of the AOT and JIT systems were analysed

```
short[] vals = new short[60];
short cnt = 0;
short average;

while(true) {
  //fake sensor sample
  setLedOn(0);
  waitMs(1);
  setLedOff(0);

  vals[cnt] = cnt;
  cnt++;

  if (cnt == 59) {
    average = 0;
    for(short s = 0; s < 60; s++) {
      average += vals[cnt];
    }
    average = (short) (average / 60);

    //fake radio transmission
    setLedOn(0);
    waitMs(1);
    setLedOff(0);

    cnt = 0;
  }
  waitMs(1000);
}
```

Figure 8.2: Implementation of an application to simulate sampling of 60 sensor readings, averaging of the readings and transmission over the air. An LED was used to represent sensor sampling and radio transmission so as to remove any unfair comparisons due to driver implementation.

in these experiments, whilst only the execution was analysed for the TakaTuka virtual machine and the reprogramming overhead was assumed to be as optimal. The execution overhead was then analysed per complete 60 second cycle (including 60 sensor samples each represented by an LED toggle, and 1 radio transmission represented by another radio toggle) and the resultant energy consumption was 326 mJ, 357 mJ and 410 mJ for AOT, JIT and interpretation using the TakaTuka virtual machine respectively.

Therefore, the lifecycle energy consumption equations can be altered to include the true analysed consumptions values for this experiment as follows:

$$E_{lifecycleint} = 197mJ + \sum_{i=0}^{n-1} 410mJ \tag{8.31}$$

$$E_{lifecycleaot} = 203mJ + \sum_{i=0}^{n-1} 326mJ \tag{8.32}$$

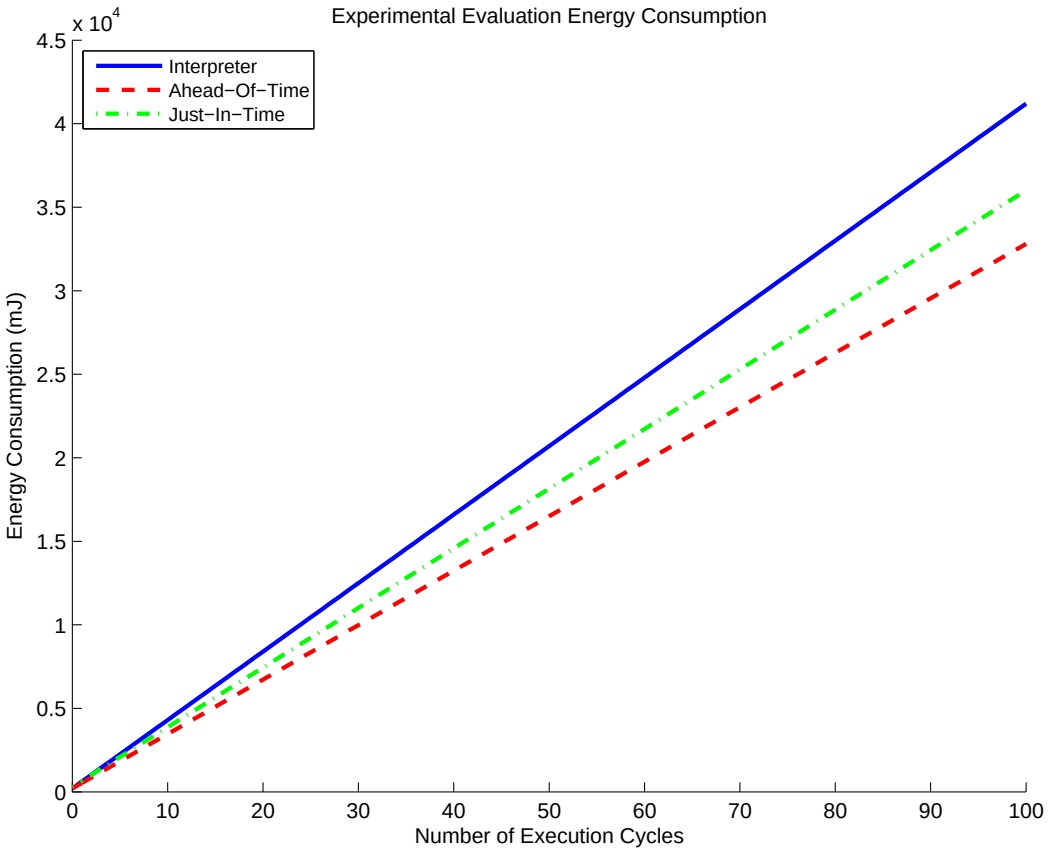

Figure 8.3: Energy consumption for reprogramming and execution of application over a given number of cycles.

$$E_{lifecyclejit} = 303mJ + \sum_{i=0}^{n-1} 357mJ \tag{8.33}$$

Equations 8.31, 8.32 and 8.33 are plotted in Figure 8.3. As can be seen from the figure and equations the energy consumed for the reprogramming and execution lifecycle is linear. As the number of cycles increases the gains achieved for using AOT over JIT and interpretation increases, and also the gains for using JIT over interpretation increases. The curve profiles tend to match the model used in 8.1 for single byte updates more than that of larger sized updates. The reason is due to the simple model used to determine the reprogramming and execution overhead based on the size of the update. However, in actual fact the resultant energy consumption will be highly dependent not only on the size of the update but also the execution profile of the application. The profile consists of different factors including the percentage of active execution and sleep states, the type of processing involved (certain instructions require more execution than others) and the duration that peripheral hardware is switched on amongst other aspects. The application used in this experiment, was purposely chosen to emphasise a typical WSN application with a high sleep percentage state and a low active execution state. As can be seen from the results, even with a high sleep profile, the energy consumed is substantially

larger for interpretation and JIT compilation than that of AOT compilation. For the given experiment, after 100 cycles interpretation requires around 26% more energy, and JIT compilation requires 10% more energy than that of AOT compilation; and the interpreted version requires 14% more energy than that of the JIT implementation. This experimental evaluation therefore shows, that even for application with high percentages of sleep periods, that both AOT and JIT compilation provides a better platform for reprogramming (and execution) than that of an interpreter.

Chapter 9

Conclusions

Programming application software for wireless sensor networks can prove to be a daunting task which is usually left to expert embedded systems programmers. High level programming languages could be used to ease the burden of programming wireless sensor networks and also increase the adoption of such technology. Recent virtual machine interpreter based initiatives to enable Java for wireless sensor nodes demonstrates the benefits of a higher level language programming paradigm for such severely resource constrained devices, however interpretation suffers from high execution overheads. Run-time compilation techniques are commonly used to increase performance. However, the general consensus in the wireless sensor networks community is that run-time compilation is impossible, impractical, complex or too resource hungry for such severely resource constrained devices (Palmer, 2004; Koshy and Pandey, 2005; Pandey and Koshy, 2006; Koshy et al., 2008; Aslam, 2011). This thesis demonstrates that Ahead-Of-Time and Just-In-Time run-time compilation techniques for severely resource constrained devices are in fact possible, practical and can be implemented in a simple manner which does not require any more memory than that of an interpreter. This concluding chapter summarises the work presented in this thesis and suggests some future directions.

9.1 Choosing the Ideal Run-time Platform

In this thesis, run-time compilation techniques were proposed for WSNs in aim of supporting efficient execution of (Java) bytecode whilst not sacrificing reprogrammability or platform independence. Alternative Java bytecode execution approaches which can be implemented on typical WSN nodes include Java-to-Native (and Java-to-C) code compilation and interpretation. No known Java-to-Native code techniques for WSN nodes were available at the time of writing and therefore native code benchmarks implemented were developed in C. Java-to-Native code is likely to be slower than that of C code due to Java abstractions. The benchmarks however served their purposed to identify the best

Table 9.1: Run-time Platform Support for Requirements

	Java-to-Native	Interpretation	AOT	JIT
Reprogramming	Low	High	High	High
Platform Independence	Low	High	High	High
Speed	High	Very Low	Medium	Low
Short Execution Cycles	High	High	High	High
Long Execution Cycles	High	Very Low	High	Low
Program Space Efficiency	Low	High	Very Low	Medium
Energy Consumption Efficiency	Very High	Low	High	Medium

case execution time for the different test cases. The Ahead-of-Time and Just-in-Time compilation techniques proposed in this thesis have been shown to provide beneficial execution efficiency and power efficiency improvements over interpretation. The scope of this thesis is primarily to demonstrate that such run-time compilation techniques are possible and practical. It is not the belief of the author that any one particular run-time platform provides a silver bullet solution to supporting Java for typical sensor nodes. Therefore, a comparison of the different approaches is provided in Table 9.1. The table highlights how the different approaches vary in support for application requirements. Values for how well the requirement is supported is categorised into Very Low, Low, Medium and High. Only the requirements where the approaches differ in their support are included in the table which are: *reprogramming*, i.e. the ability to be able to reprogram a sensor node over the air; *platform independence* defines whether the platform can support types of sensor nodes; *speed* is how fast the platform can execute code; *short execution cycles* and *long execution cycles* represents how well the platform is suited for applications that have short or long execution cycles; and *program space* defines how well the platform caters for minimising program space usage. The ideal platform for a particular application can therefore be selected by reviewing each requirement's criticality and eliminating any platforms that do not support it.

In respect to reprogramming Java-to-Native code compilation provides little support since code is typically statically linked and more so since native code tends to be larger than bytecode, higher transmission costs of code will be incurred. The Interpretation, AOT and JIT platforms are all well suited to support reprogramming and also platform independence primarily due to the bytecode encoding. Java-to-Native code compilation does not provide good support for platform independence in that if a network were to consist of different types of sensor nodes, different code updates would have to be sent to the different types of nodes (since the different types of sensor nodes support different native code instruction sets). As noted from our experimental evaluation of execution speeds, Interpretation suffers greatly with very low execution speeds, JIT offers a slightly better speed improvement, AOT a decent speed and as expected native code offers the greatest speed efficiency. Although it was shown the interpreter's performance was slow even for short execution cycles, it should be possible to speed it up by offloading more execution into the natively written drivers and libraries. Therefore, an interpreter should

be possible to suitably support those applications with short execution cycles and long sleep periods. For long execution cycles (and short sleep periods), as described in Chapter 3, an interpreted platform will not be able to cope with the quality of service required, whilst a JIT platform provides slightly better support. On to a requirement where an interpretation platform greatly beats other platforms is that of program space requirements. Although, in our evaluation the interpreted version actually results in program sizes comparable to that of the other approaches, bytecode intended for interpretation can actually be much smaller as explained in Chapter 5. The JIT platform proposed requires slightly more space (due to basic block identification bytecode instructions). Native code would tend to be larger, and therefore program space is not well supported by a Java-to-Native code compiler (without compiling out Java abstractions), and the AOT compiler performs worst when it comes to program space requirements. This is due to the simple compilation process used. Native code generated on a development machine will undoubtedly result in the most energy efficient platform (for application execution). As shown in Chapter 3, the slower code executes, the slower it takes to turn on and off hardware. Indirect energy consumption costs will also be incurred, and therefore the energy consumed is greatly affected by the platform's speed.

9.2 Summary of Work

The aim of this thesis has been to investigate whether the preconception that run-time compilation techniques are in fact something beyond the power and resources of severely resource constrained devices commonly used in wireless sensor networks. Run-time compilation techniques were designed and implemented to establish whether the claims are indeed true. This thesis describes techniques proposed to enable run-time compilation for resource constrained devices. Following is a summary of the work presented in this thesis leading to a conclusion in relation to whether run-time compilation techniques are in fact applicable for wireless sensor networks.

Chapter 1 provided a discussion on the current programming environments used for wireless sensor network development and the need for higher level abstractions to decrease development costs and increase adoption of the technology. Recent initiatives to enable higher level abstractions by means of a Java interpreter were introduced and the high execution costs associated with such an approach were discussed. The consensus in the wireless sensor network field concerning the impossibility and impracticality of run-time compilation for resource constrained devices is mentioned and thus the problem statement is introduced, that is, whether the consensus is preconceived. The challenge, as indicated by the assumption that run-time compilation is not for WSN class devices, is identified as the limited program and memory resources. An introduction to motivation behind the work was then provided, along with how the work relates to existing approaches.

Having introduced the problem and main question surrounding the thesis, in Chapter 2, an introduction to related work and an overview of background areas is provided. Main wireless sensor networks requirements and issues are highlighted with further focus on WSN programming paradigms. The work presented in this thesis focuses on a Java programming paradigm and therefore further background on the Java virtual machine and bytecode is presented followed by an analysis of the desirability and undesirability of Java for WSNs and a discussion highlighting Java features that were not designed for WSNs. Techniques suggested by recent initiatives to enable Java programming environments for WSNs are then discussed and followed by techniques proposed to increase Java execution performance including run-compilation techniques used in traditional computing platforms.

To justify the work involved in implementing run-time compilation of bytecode in wireless sensor networks, an analysis of execution overheads for both a native code and interpreter execution platform was performed and presented in Chapter 3. An estimation of the expected lifetime of a device for varying active and sleep durations for natively compiled firmware is given in comparison to an interpreted version for code that performs the same task. The estimation shows that interpretation results in a substantial decrease in expected lifetime even for low duty cycles. Since, wireless sensor networks do not just consist of computation but also sensing and radio communication, the lifetime model was further expanded to include an element of sensor sampling and radio transmission. The model is based upon the assumption that a single sensor sample and radio transmission is made per active and sleep period. The analysis shows that even low active periods and long sleep periods results in a substantial decrease in expected lifetime for interpreted code with a single sensor sample and radio transmission. The general approach used in WSN application development advocates higher computation in order to minimise transmission. When this approach is kept in mind the gains of using a native code paradigm over an interpreted one are further amplified. Further to an initial analysis by means of estimating execution overheads, an experimental analysis of an existing virtual machine was performed which highlighted indirect overheads inherent in interpretation. The indirect overheads are due to slower execution speeds which results in peripheral hardware being on for longer periods, and therefore consuming more energy. This chapter provided enough motivation to continue investigating whether run-time compilation techniques can be achieved.

Following the motivation behind the continuation of pursuing run-time compilation techniques for wireless sensor networks, design requirements and choices for an Ahead-Of-Time compilation scheme are then presented in chapter 4. A simple run-time compilation mechanism is used to convert bytecode into native code by mimicking the Java operand stack by natively pushing and popping to a stack. Simple optimizations on the resultant generated code are further proposed to increase performance and decrease resultant

application size. To overcome the memory constraints imposed in WSNs, Gradual Compilation was proposed so that code can be gradually compiled as it is received without having to receive a whole function's code before beginning compilation. Although, Java is meant to relieve the application developer of lower level details it was decided to still expose such lower level registers and interrupts in the event that the developer required to use such mechanisms. A novel technique to expose registers and interrupts to Java is thereby provided.

An evaluation of the Ahead-Of-Time compilation method proposed was presented in chapter 5. Results show that the proposed AOT compilation method achieves significantly faster execution of code compared with interpretation, which in turn results in an increased lifetime. A comparison of program encoding size is then presented whereas it is shown that the bytecode produced for the proposed AOT compilation scheme is less than that of native code generated from C for most cases and always less than that of an interpreter (although, the bytecode used in the interpreter should in general be smaller or equivalent to the bytecode proposed due to gradual compilation). The generated native code for the AOT compilation method proposed is also for most test applications smaller than that of the interpreter (although this most definitely should not be the case). Therefore, it was shown that as regards to application logic size (for both the bytecode and native code encodings), the AOT compilation scheme results in comparative sizes to that of an interpreter. Furthermore, the complete size of the underlying virtual machine along with the application logic on top of it was analysed. The results show that the AOT compiler footprint along with the application logic resulted in comparable or smaller sizes to that of the interpreter. Therefore, it was confirmed that the AOT compilation scheme achieves similar program space requirements to the interpreter and therefore claims that run-time compilation is impossible or impractical on such devices is rejected, or at least that the same applies to interpreters.

Although the evaluation presented in chapter 5 demonstrates that the total application size for AOT compilation is comparable to an interpreter, further work into minimising the application size was performed by designing and implementing Just-In-Time compilation for resource constrained devices as described in chapter 6. The challenge in doing so, as previously mentioned, is the program and memory space constraints. Writing to program flash consumes substantial energy and therefore it is proposed to compile and execute native code from RAM. To enable JIT compilation for such resource constrained devices Basic Block JIT compilation was proposed. By JIT compiling code at the granularity of basic blocks, the memory required to store and execute the generated native code can be drastically reduced. More so, code blocks that are not executed due to conditional statements not being evaluated will not be compiled and thus speed up the compilation process. To further facilitate JIT compilation for resource constrained devices it was proposed to perform Offline Basic Block Analysis on a host machine prior to transmitting bytecode to the sensor node, and therefore the sensor node would be

relieved of basic block analysis and can concentrate purely on executing the code. To further support efficient execution of code and minimise the JIT compiler footprint, direct native code calls to the JIT compiler are generated and stored before each basic block of bytecode. In doing so, by exploiting the underlying hardware architecture, the JIT compiler can establish the location of the bytecode to be compiled and then return to the next basic block to be executed. To further support a lightweight JIT compilation scheme, a circular JIT cache is used to store previously generated native code and then when required to discard native code due to size limitations, the oldest native code can be discarded from the cache. The simple JIT cache will ensure that at least execution of loops will exhibit optimised behaviour.

An evaluation of the JIT compilation scheme is then provided in chapter 7. It is shown that JIT compilation can be achieved with less of a footprint than an interpreter achieving comparable or less execution speeds.

Reprogrammability of bytecode is advantageous due to its platform independence and also bytecode encoding which usually provides a smaller encoding size. In chapter 8 it is shown that the overheads of reprogramming sensor nodes which utilise a run-time compilation scheme result in substantially less overheads than if interpretation is used.

Furthermore, to reiterate the main contribution of this work, the thesis that run-time compilation is possible, practical and can be implemented on severely resource constrained devices by using simple techniques which results in resource usage comparable (or better) to that of an interpreter has been shown to be true. The run-time compilation source is available from: http://sourceforge.net/projects/micrortc/

9.3 Future Work

The major part of this thesis has been concerned with determining whether run-time compilation is in fact beyond the power of severely resource constrained devices such as those commonly used in wireless sensor networks. In investigating this preconception, techniques were proposed to enable Ahead-Of-Time and Just-In-Time compilation for resource constrained devices. The proposed techniques enable run-time compilation for resource constrained devices with space requirements comparable (or smaller) than that of an interpreter, furthermore it results in a significantly more efficient execution platform. Having proven that run-time compilation is in fact possible and having provided the first AOT and JIT compilers for such resource limited devices, this thesis paves the way for further research into resource constrained run-time compilation research. In this section areas of interest are identified which could be further investigated to extend the work presented in this thesis.

9.3.1 Mixing AOT and JIT Compilation

The work presented in this thesis demonstrates that both AOT and JIT compilation can be achieved for severely resource constrained devices. It would be useful to use AOT compilation for code that is executed frequently and JIT compilation for code that is less frequently executed (as done by other traditional dynamic compilation systems). In this thesis focus was on the enabling methods to achieve run-time compilation for severely resource constrained devices. The study of enabling a dynamic compilation system for severely resource constrained devices would entail a thesis of its own.

Since the AOT and JIT compilers use the same translation logic (bar some minor additional JIT logic) the two compilation techniques can be used together with minor additional changes. Also since the AOT and JIT generated native code does not rely on an underlying run-time system to execute the code (since it is executed natively), code blocks can be seamlessly executed as Ahead-Of-Time compiled code or Just-In-Time compiled code (even within the same function). That said, due to program space constraints this would be a challenging task for an interpreter based approach since the bytecode to native code transition logic would require to be implemented on top of the interpreter logic which would result roughly in double the size of the interpreter. Therefore research should be conducted in aim of providing a hybrid AOT and JIT compilation (and execution) platform. The simplest way of enabling this would be to allow the developer to determine whether a function should be marked for AOT or JIT compilation. However, in aim of decreasing effort on behalf of the programmer, it would be interesting to develop policies to determine whether blocks of code should be marked for AOT or JIT compilation. Research into such policies should both focus on offline analysis which would take place on a host machine when converting Java bytecode into the system's intermediate bytecode, and also run-time policies that would be able to decide if any code blocks marked for JIT compilation should be permanently stored in flash (and thereafter never require any further compilation).

9.3.2 Bytecode Optimization

Bytecode optimization is an area of interest to all types of virtual machines and extensive research has been presented for bytecode optimization on traditional computing platforms. Aslam (2011) presented extensive research on bytecode compaction in their work. Positive results are demonstrated for run-time compilation in this thesis, even when compared with an interpreter which makes use of bytecode optimization. Research into offline bytecode optimization and its relationship to run-time compilation could substantially further increase the performance gains and minimise storage overhead.

9.3.3 Native Code Optimization

Bytecode to native code translation techniques that mimic the Java operand stack were proposed in this thesis to enable a simple compilation framework. Furthermore simple optimization on the generated native code were also proposed. Work on further optimizations should be investigated in aim of increasing the execution efficiency and reducing the storage requirements.

9.3.4 JIT Cache Policies

In this thesis a simple circular cache was proposed to store native code generated by the JIT compiler. Further research should be executed to determine other caching policies that may increase cache hits and minimise storage requirements for native code that is unlikely to be executed.

9.3.5 Flash Memory Management

The JIT compilation scheme proposed is supported by a circular JIT cache which stores previously generated native code in RAM (since flash writing is time and power consuming). RAM is highly constrained and therefore the JIT cache size will determine the efficiency of code intended for JIT compilation and execution. Often the flash memory available for program space is never completely used. Therefore, it would be beneficial to develop a flash memory management layer that would allow the JIT compiler to store generated native code to unused flash. Thereafter if flash memory is required by the application or system level logic, flash memory in use by JIT could be freed and if the JIT generated native code is ever required again it could be either written to other available flash memory areas, or else to a JIT cache in RAM. That said, policies to ensure that flash writing is minimised and code blocks that are more frequently executed are given priority for storage in flash would have to be investigated.

9.3.6 Integratation of the JIT Cache with the Memory Manager and Garbage Collector

Similar to the idea proposed above, although RAM is limited on such devices, this does not necessarily mean that majority of the RAM is in use for applications. The Java paradigm uses a garbage collector which releases developers from having to implement memory management. The garbage collector will release resources that are no longer in use by the application. Therefore, it is possible that substantial memory previously used by application logic is tied up waiting to be garbage collected. More so, memory usage of specific applications may not be large and therefore the JIT cache would be able to

use more memory for such applications. Hence, further research into integration of the JIT cache with the memory manager and garbage collector should be investigated so as to optimise the JIT cache size availability at run-time.

9.3.7 Debugging via Reverse Translation

Source level debugging is a useful tool to analyse code and fix bugs. In aim of supporting source level debugging of firmware loaded on a sensor node, further work should be performed in reverse translation from native code to the original Java source. This would involve the inverse of the compilation process described in chapter 4, and therefore being native code to intermediate bytecode, which is further translated to Java bytecode and then can be matched to Java source files and lines.

9.3.8 Other Language and Bytecode Alternatives

Java was used as the programming language of choice for this work due to its popularity and other recent Java interpreter based virtual machines for WSNs. However, Java may or may not be the ideal language to use for enabling development of WSN applications. Although the question of which language would best suit development is not the scope of the thesis, it is definitely an area of interest. When considering the different languages it is important to keep in mind the underlying program encoding (be it native code, bytecode or other) and how such encodings effect platform dependency, update sizes, execution efficiency, storage requirements and reprogrammability.

Appendix A

Bytecode to Native Code Translations

A.1 Load and Store Instructions

A.1.1 Local Variable Value Loading

Bytecode Instructions	`fload`, `fload_0`, `fload_1`, `fload_2`, `fload_3`, `iload`, `iload_0`, `iload_1`, `iload_2`, `iload_3`
Stack [before] ⇒ [after]	*...* ⇒ *..., value*
Native Code Translation	`PUSH <VARIABLE LSW OFFSET>(R4)` `PUSH <VARIABLE MSW OFFSET>(R4)`

Description

The bytecode instructions are used to push local variable `float` and `int` (32-bit) values onto the stack. The translation therefore pushes the least significant word (16-bits in the case of the MSP430 microcontroller used) onto the native microcontroller stack, followed by pushing the most significant word onto the native microcontroller stack.

The various `fload_<n>` and `iload_<n>` bytecode instructions implicitly state the local variable index number, whilst it is provided as a parameter for `fload` and `iload`. The compiler must therefore prior to this, calculate the local variable's position in respect to the current method stack frame.

A.1.2 Local Variable Reference Loading

Bytecode Instructions	`aload`, `aload_0`, `aload_1`, `aload_2`, `aload_3`
Stack [before] ⇒ [after]	*...* ⇒ *..., objectref*
Native Code Translation	`INCD.W R8` `MOV.W <REFERENCE OFFSET>(R4), 0x0000(R8)`

Description

The bytecode instructions are used to push local variable references onto the (reference) stack. The run-time compiler implementation separates the value stack from the operand stack, and the microcontroller's native stack is used to represent the value stack. Therefore, the reference stack requires a software implemented stack. The `R8` register is used to represent the reference stack pointer. The native code translation increments the reference stack pointer, and thereafter copies the object reference from the local variable method frame to the address of the reference stack pointer. The compiler must prior to this calculate the object reference position offset in respect to the current method stack frame.

The local variable reference position is implicitly stated in the `aload_<n>` instructions whilst the position is provided as a parameter for the `aload` bytecode instruction.

A.1.3 Local Variable Value Storing

Bytecode Instructions	`fstore`, `fstore_0`, `fstore_1`, `fstore_2`, `fstore_3`, `istore`, `istore_0`, `istore_1`, `istore_2`, `istore_3`
Stack [before] ⇒ [after]	*..., value* ⇒ *...*

Native Code Translation	`POP R5`
	`MOV.W R5, <VARIABLE MSW OFFSET>(R4)`
	`POP R7`
	`MOV.W R7, <VARIABLE LSW OFFSET>(R4)`

Description

The bytecode instructions are used to store `float` and `int` (32-bit) values stored on the operand stack into a local variable. The translation involves popping the most significant word (16-bits) from the native microcontroller stack, and thereafter storing the popped value to the position of the variable's most significant word. The same procedure is followed for the least significant word.

The various `fstore_<n>` and `istore_<n>` bytecode instructions implicitly state the local variable index number, whilst it is provided as a parameter for `fstore` and `istore`. The compiler must therefore prior to this, calculate the local variable's position in respect to the current method stack frame.

A.1.4 Local Variable Reference Storing

Bytecode Instructions	`astore`, `astore_0`, `astore_1`, `astore_2`, `astore_3`
Stack [before] ⇒ [after]	*..., objectref* ⇒ *...*

Native Code Translation	`MOV.W @R8, R5`
	`DECD.W R8`
	`MOV.W R5, <REFERENCE OFFSET>(R4)`

Description

The bytecode instructions are used to store object references stored on the (reference) stack into a local reference variable. The `R8` register is used as a software implemented reference stack. The native code translation copies the reference on the top of the reference stack into an intermediate register (`R5`), decrements the reference stack (which represents the popping of the top reference), and then copies the reference stored in the intermediate register to the local variable reference position in the method frame.

The local variable reference position is implicitly stated in the `astore_<n>` instructions whilst the position is provided as a parameter for the `astore` bytecode instruction.

A.1.5 Value Constant Loading

Bytecode Instructions	`bipush`, `fconst_0`, `fconst_1`, `fconst_2` `iconst_m1`, `iconst_0`, `iconst_1`, `iconst_2`, `iconst_3`, `iconst_4`, `iconst_5`, `sipush`
Stack [before] ⇒ [after]	*...* ⇒ *..., value*
Native Code Translation	`PUSH <VALUE LSW>` `PUSH <VALUE MSW>`

Description

The bytecode instructions are used to push constant `byte`, `float`, `int` and `short` values onto the stack as 32-bit values. The native code translation pushes the value's least significant word (16-bits) followed by the most significant word onto the native microcontroller stack.

The `fconst_<n>`, `iconst_<n>` and `iconst_m1` (minus 1) bytecode instructions implicitly state the constant value, whilst the constant value is provided as a parameter for `bipush` and `sipush`.

A.1.6 Reference Constant Loading

Bytecode Instruction	`aconst_null`
Stack [before] ⇒ [after]	*... ⇒ ..., null*
Native Code Translation	`INCD.W R8` `CLR.W 0x0000(8)`

Description

The bytecode instructions is used to push the `null` reference value onto the (reference) stack. The `null` reference is represented as the binary value 0. The `R8` register is used as a software implemented reference stack. The native code translation increments the reference stack pointer, and thereafter clears the value stored in the address pointed by the reference stack pointer (or rather sets the value to binary 0).

A.2 Arithmetic Instructions

A.2.1 Natively Supported Dual Stack Operand Integer Arithmetic Translations

Bytecode Instructions `iadd`, `iand`, `ior`, `isub`, `ixor`

Stack [before] ⇒ [after] *..., value1, value2 ⇒ ..., result*

Native Code Translation

```
POP R15
POP R14
POP R13
POP R12
<OPERATION 1> R14, R12
<OPERATION 2> R15, R13
PUSH R12
PUSH R13
```

Description

The bytecode instructions are used to perform integer arithmetic operations on values stored on the stack, and thereafter push the result back on the stack. The `iadd` (integer addition), `iand` (integer bitwise and), `ior` (integer bitwise or), `isub` (integer subtraction) and `ixor` (integer bitwise xor) bytecode instructions can be implemented using native microcontroller operations. The native code translation first pops the second value's most significant word (16 bits) into a working register followed by the least significant word. The first value's most significant word and least significant word are then also popped to working registers. Then a first operation is performed on the least significant words of the second (R14) and first value (R12) (and the result will be stored in R12). Then a second operation is performed on the most significant words of the second (R15) and first value (R13) (and the result will be stored in R13). The reason why the least significant word is processed first is due to carrying over bits for addition and subtraction operations. Finally, the results are put back on the stack by pushing the resultant least signficant word (R12) and most significant word (R13).

The native `ADD.W` and `ADDC.W` instructions are used for integer addition; and `SUB.W` and `SUBC.W` for integer subtraction. The same native instruction can be used for `<OPERATION 1>` and `<OPERATION 2>` for integer bitwise and, or and xor. The native instructions are `AND.W`, `BIS.W` and `XOR.W` respectively.

A.2.2 Other Natively Supported Dual Operand Integer Arithmetic Translations

Bytecode Instruction `iinc`

Stack [before] ⇒ [after] ... ⇒ ...

Native Code Translation

```
ADD.W <CONST>,<VARIABLE LSW OFFSET>(R4)
ADC.W <VARIABLE MSW OFFSET>(R4)
```

Description

The bytecode instruction is used to increment the value of a local variable by a constant specified byte value, which is provided as a parameter to the instruction. The native translation first performs an addition of the constant value and the variable's least significant word (16-bits). Thereafter, if the previous operation resulted in a carry the most significant word will be increased by 1 by using the *add carry to destination* native instruction.

A.2.3 Software Supported Dual Operand Arithmetic Translations

Bytecode Instructions	`fadd`, `fdiv`, `fcmpg`, `fcmpl`, `fmul`, `fsub`, `idiv`, `imul`, `ishl`, `ishr`, `iushr`
Stack [before] ⇒ [after]	*..., value1, value2* ⇒ *..., result*
Native Code Translation	`CALL <OPERATION FUNCTION>`

Description

The underlying microcontoller (as most other processors) does not provide native instructions to facilitate native execution of all possible arithmetic bytecode instructions. Therefore, the arithmetic has to be implemented in software. Therefore, functions were implemented in software which pop the required values from the stack, perform the required operation and store the result on the stack. The functions have to manipulate the microcontroller stack to ensure that execution is returned to the caller, since the microcontroller stack is used both for execution control as well as for the value operand stack. The following template code is used for each operation:

```
__<OPERATION_FUNCTION_NAME>
  POP R10 ;pop the return address of the caller into R10
  POP R15 ;pop value2 MSW into R15
  POP R14 ;pop value2 LSW into R14
  POP R13 ;pop value1 MSW into R13
  POP R12 ;pop value1 LSW into R12

  <PERFORM OPERATION HERE>

  PUSH <LSW RESULT> ;usually use R12
  PUSH <MSW RESULT> ;usually use R13

  MOV.W R10, PC
```

The code first pops the return address of the caller function into `R10`. This is followed by popping the second and first values' most significant words and least significant words into working registers. The arithmetic operation is then performed. The resultant least significant word and most significant word is then pushed back on the stack. Execution is then passed back to the the caller by setting the value of the program counter to the return address (stored in `R10`).

A.2.4 Software Supported Single Operand Arithmetic Translations

Bytecode Instructions `fneg`, `ineg`

Stack [before] ⇒ [after] *..., value ⇒ ..., result*

Native Code Translation `CALL <OPERATION FUNCTION>`

Description

A similar approach is used for single operand arithmetic translations to that of software supported dual operand translations. The native code generated for the operation is a `CALL` instruction to a function that implements the single operand arithmetic operation. The function template for single operand instructions only differs slightly to the dual operand function template, as can be seen below:

```
__<OPERATION_FUNCTION_NAME>
  POP R10 ;pop the return address of the caller into R10
  POP R15 ;pop value MSW into R15
  POP R14 ;pop value LSW into R14

  <PERFORM OPERATION HERE>

  PUSH R14 ;push result LSW
  PUSH R15 ;push result MSW

  MOV.W R10, PC
```

The return address of the caller is popped into `R10`. Then, only the most significant word and least significant word for a single value on the stack need to be popped into working registers. The single operand operation is performed, and thereafter the result least significant word and most significant word (stored in the same registers) is pushed on the stack. Execution is returned to the caller by changing the value of the program counter to that of the return address (previously stored in `R10`.

A.3 Type Conversion Instructions

A.3.1 Software Supported Type Conversion Transformations

Bytecode Instructions `f2i`, `i2f`

Stack [before] ⇒ [after] *..., value ⇒ ..., result*

Native Code Translation `CALL <OPERATION FUNCTION>`

Description

The bytecode instructions provided conversions from `float` to `integer` (`f2i`) and vice versa (`i2f`). Conversions between `float` and `integer` values are not natively supported by the underlying microcontroller instruction set. Therefore, conversions are provided by software implementations. The native code translation only requires to provide a native `CALL` instruction to a function that implements the conversion. The function template is as follows:

```
__<OPERATION_FUNCTION_NAME>
  POP R10 ;pop the return address of the caller into R10
  POP R15 ;pop value MSW into R15
  POP R14 ;pop value LSW into R14

  <PERFORM OPERATION HERE>

  PUSH <LSW CONVERTED VALUE>
  PUSH <MSW CONVERTED VALUE>

  MOV.W R10, PC
```

The return address of the caller is popped into `R10`. Then, the most significant word and least significant word for the value being converted is popped from the stack into working registers. The conversion is performed, and thereafter the result least significant word and most significant word is pushed on the stack. Execution is returned to the caller by changing the value of the program counter to that of the return address (previously stored in `R10`.

A.3.2 Natively Supported Type Conversion Transformations

Bytecode Instruction `i2s`

Stack [before] ⇒ [after] *..., value ⇒ ..., result*

Native Code Translation POP R5

Description

The `i2s` bytecode instruction is used to convert an integer value to a short value. The native code translation only requires the most significant word to be popped from the stack.

A.4 Object Creation and Manipulation

A.4.1 Object Instantiation

Bytecode Instruction `new`

Stack [before] ⇒ [after] *... ⇒ ..., objectref*

Native Code Translation

```
MOV.W <SIZE>,R12
MOV.W <CLASS_ID>,R5
CALL __DoNewObject
```

Description

The `new` bytecode instruction is used to create an instance of the class specified as a parameter to the instruction. The native code generated first stores the oject size into `R12`, then the class numeric ID into `R5`. This is followed by a `CALL` to a function that performs the new object instantiation. The object will then be created inside `__DoNewObject` and placed on the reference stack.

A.4.2 Array Creation

Bytecode Instructions `anewarray, newarray`

Stack [before] ⇒ [after] *..., count ⇒ ..., arrayref*

Native Code Translation

```
MOV.W <CLASS_ID_OR_ARRAY_TYPE>, R7
CALL __DoNewArray
```

Description

The `anewarray` and `newarray` bytecode instructions are used to create an array of objects or values respectively. In the case of an object array creation, the object class is specified as a parameter to the instruction; whilst the array type is specified for an array consisting of primitive value types. The native code translation copies the class numeric ID or the array type to `R7`, and then calls a function implemented in the run-time system to create the array. `__DoNewArray` will create an array of type specified by `R7` with its size specified on the operand stack.

A.4.3 Instance Field Retrieval

Bytecode Instruction `getfield`

Stack [before] ⇒ [after] *..., objectref* ⇒ *..., value*

Native Code Translation

```
MOV.W @R8,R6
DECD.W R8
PUSH <FIELD LSW OFFSET>R6
PUSH <FIELD MSW OFFSET>R6
```

Description

The `getfield` bytecode instruction is used to retrieve an instance field value for an object placed on the (reference) stack. The field index is specified as a parameter to the instruction. The native code translation first *pops* the object reference on the reference stack into `R6` (by copying the value from the reference stack, and decrementing the reference stack pointer). Thereafter, the least and most significant words of the field value is pushed on to the stack. The compiler therefore requires to calculate the index of the field. The code generated in the code above is for that of a 32-bit field.

A.4.4 Static Field Retrieval

Bytecode Instruction	`getstatic`
Stack [before] ⇒ [after]	*... ⇒ ..., value*

Native Code Translation	`MOV.W <STATIC FIELD LSW ADDRESS>, R5`
	`PUSH @R5`
	`MOV.W <STATIC FIELD MSW ADDRESS>, R6`
	`PUSH @R6`

Description

The `getstatic` bytecode instruction is used to retrieve a static field value. The field index is specified as a parameter to the instruction. Since static field locations are known at (run-)compile time, the native code translation only requires to push the values at the static field location onto the stack. Therefore, the translation copies the field's least and most significant word addresses into working registers and thereafter pushes the contents of the addresses onto the stack. The code generated in the code above is for that of a 32-bit field.

A.4.5 Instance Field Storing

Bytecode Instruction `putfield`

Stack [before] ⇒ [after] *..., objectref, value ⇒ ...*

Native Code Translation

```
MOV.W @R8, R6
DECD.W R8
POP R5
MOV.W R5, <FIELD MSW OFFSET>(R6)
POP R5
MOV.W R5, <FIELD LSW OFFSET>(R6)
```

Description

The `putfield` bytecode instruction is used to store a value (on the stack) in an instance field. The field index is specified as a parameter to the instruction. The native code translation first *pops* the object reference on the reference stack into `R6` (by copying the value from the reference stack, and decrementing the reference stack pointer). Thereafter, the value's most significant word is popped off the stack and copied to instance field most significant word. The same is done for the least significant word. During code generation, the field offset is calculated by the compiler. The code generated in the code above is for that of a 32-bit field.

A.4.6 Static Field Storing

Bytecode Instruction `putstatic`

Stack [before] ⇒ [after] *..., value ⇒ ...*

Native Code Translation

```
POP R5
MOV.W R5, <STATIC FIELD MSW ADDRESS>
POP R5
MOV.W R5, <STATIC FIELD LSW ADDRESS>
```

Description

The `putstatic` bytecode instruction is used to store a value (on the stack) in a static field. The field index is specified as a parameter to the instruction. Since static field locations are known at (run-)compile time, the native code translation only requires to pop the value off the stack and thereafter copy the value to the static field location. Therefore, the translation pops the value's most significant word and then copies it to the static field's most significant word's address. The same is done for the least significant word. The code generated in the code above is for that of a 32-bit field.

A.4.7 Array Value Loading

Bytecode Instructions `baload`, `faload`, `iaload`, `saload`

Stack [before] ⇒ [after] *..., arrayref, index ⇒ ..., value*

Native Code Translation

```
POP R5
MOV.W @R8, R6
DECD.W R8
INCD.W R6
INCD.W R6
RLA.W R5
RLA.W R5
ADD.W R5, R6
PUSH 0x0000(R6)
PUSH 0x0002(R6)
```

Description

The bytecode instructions are used to push an array element value onto the operand stack. The native code translation pops the index from the operand stack, and the pops the array reference from the (reference) stack (by copying the value from the reference stack, and decrementing the reference stack pointer). The array pointer then is incremented to skip array information including the array type and array size. The index specified is an array index number, therefore the `R5` register is shifted right twice so that the index number is translated to the byte offset. The offset is then added to the array pointer, `R6`, to point to the element of interest. The least and most significant words of the array element value are pushed onto the stack. The code generated in the code above is for that of a 32-bit field.

A.4.8 Array Value Storing

Bytecode Instructions `bastore`, `fastore`, `iastore`, `sastore`

Stack [before] ⇒ [after] *..., arrayref, index, value ⇒ ...*

Native Code Translation

```
POP R11
POP R5
POP R6
MOV.W @R8, R7
DECD.W R8
INCD.W R7
INCD.W R7
RLA.W R6
RLA.W R6
ADD.W R6, R7
MOV.W R5, 0x0000(R7)
MOV.W R11, 0x0002(R7)
```

Description

The bytecode instructions are used to store a value placed on the stack into an array element. The native code translation first pops the most and least significant words. Then, similar to the array element loading operation, it pops the index from the operand stack, and the pops the array reference from the (reference) stack (by copying the value from the reference stack, and decrementing the reference stack pointer). The array pointer then is incremented to skip array information including the array type and array size. The index specified is an array index number, therefore the `R6` register is shifted right twice so that the index number is translated to the byte offset. The offset is then added to the array pointer, `R7`, to point to the element of interest. The least and most significant words of the array element are set to the values previously popped from the stack. The code generated in the code above is for that of a 32-bit field.

A.4.9 Array Length Retrieval

Bytecode Instruction	`arraylength`
Stack [before] ⇒ [after]	*..., arrayref* ⇒ *..., length*

Native Code Translation	`MOV.W @R8, R5`
	`DECD.W R8`
	`PUSH 0x0002(R5)`

Description

The `arraylength` bytecode instruction is used to retrieve an array's length and placing it on the operand stack. The native code translation involves getting a pointer to the array specified by the array reference on the stack (by copying the value from the reference stack, and decrementing the reference stack pointer). Then, the array length can be pushed onto the stack.

A.5 Control Transfer Instructions

A.5.1 Patching In Jump Addresses

Execution can be transferred based on unconditional jumps and also conditional statements. In order to perform a single pass of bytecode, the compiler leaves jump destinations empty and then fills them in once the jump native code location can be determined.

A.5.2 Unconditional Jumps

Bytecode Instruction `goto`

Stack [before] ⇒ [after] *... ⇒ ...*

Native Code Translation `BR <DESTINATION>`

Description

The `goto` bytecode instruction is used to unconditionally jump to a different bytecode location. The run-time compiler upon encountering a `goto` bytecode instruction will generate a branch instruction and leave the native code destination blank. Once the destination address can be determined it will then be patched in.

A.5.3 Value Based Conditional Instructions

Bytecode Instructions `if_icmpeq`, `if_icmpge`, `if_icmpgt`, `if_icmple`, `if_icmplt`, `if_icmpne`

Stack [before] ⇒ [after] *..., value1, value2 ⇒ ...*

Native Code Translation

```
POP R6
POP R5
POP R15
POP R14
CMP.W R6, R15
<CONDITIONAL JUMP> <DESTINATION>
CMP.W R14, R5
<CONDITIONAL JUMP> <DESTINATION>
```

Description

The bytecode instructions are used to jump to a bytecode instruction address if a comparison between two values on the stack holds for a particular condition. The native code translation first pops the most and least significant words of *value2*, followed by *value1*. Thereafter, the most significant words are compared against eachother. Then if the particular condition holds, execution is transferred to the native code destination address. If the condition does not hold, then the least significant words are compared against and execution is transferred to the destination address if the comparison holds. Otherwise, execution continues from the next native code instruction. The code generated in the code above is for that of an `int` datatype.

A.5.4 Conditional Instructions Based on Comparison with Zero

Bytecode Instructions `ifeq`, `ifge`, `ifgt`, `ifle`, `iflt`, `ifne`

Stack [before] ⇒ [after] *..., value ⇒ ...*

Native Code Translation

```
POP R6
POP R15
CMP.W R6, #0
<CONDITIONAL JUMP> <DESTINATION>
CMP.W R14, #0
<CONDITIONAL JUMP> <DESTINATION>
```

Description

The bytecode instructions are used to jump to a bytecode instruction address if a comparison between a value on the stack and zero holds for a particular condition. The native code translation pops the most and least significant words of the *value*. Thereafter, the most significant word is compared against zero. Then if the particular condition holds, execution is transferred to the native code destination address. If the condition does not hold, then the least significant word is compared against zero and execution is transferred to the destination address if the comparison holds. Otherwise, execution continues from the next native code instruction. The code generated in the code above is for that of an `int` datatype.

A.5.5 Reference Based Conditional Instructions

Bytecode Instructions	`ifnonnull`, `ifnull`
Stack [before] ⇒ [after]	*..., objectref* ⇒ *...*

Native Code Translation	`MOV.W @R8, R5`
	`DECD.W R8`
	`CMP.W R5, #0`
	`<CONDITIONAL JUMP> <DESTINATION>`

Description

The bytecode instructions are used to compare whether a reference is non-null (`ifnonnull`) or null (`null`). The native code translaton first pops the object reference off of the (reference) stack (by copying the value from the reference stack, and decrementing the reference stack pointer). The pointer value is then compared to zero, or null, and if the comparison holds for the particular condition, execution is transferred to the native code destination address. Otherwise, execution continues at the next native code instruction.

A.6 Method Invocation and Return Instructions

A.6.1 Static Method Invocation

Bytecode Instruction `invokestatic`

Stack [before] ⇒ [after] *..., [arg1, [arg2 ...]] ⇒ ...*

Native Code Translation
```
CALL <STATIC METHOD ADDRESS>
```

Description

The `invokestatic` bytecode instruction is used to call static methods. The class and method identification is passed as a parameter to the instruction. The compiler will first find the address of the static method, and then generate the native code translation which results in a native `CALL` instruction to the static method.

A.6.2 Instance Method Invocation

Bytecode Instructions `invokeinterface`, `invokespecial`, `invokespecial`

Stack [before] ⇒ [after] *..., objectref, [arg1, [arg2 ...]] ⇒ ...*

Native Code Translation
```
MOV.W <CLASS_ID>, R12
MOV.W <METHOD_ID>, R13
CALL <EXECUTION HANDLER>
```

Description

The bytecode instructions are used to invoke instance methods. The class and method identification is pass as a parameter to the instruction. An instance method call cannot be statically linked at (run-time) compile time, since the rules of method invocation depend on the type of the object and not the type of class variable. Therefore, a run-time environment execution handler is used to find the correct method to invoke. Thus, the class ID and method ID are copied to working registers, and thereafter the execution handler can correctly identify the correct method to invoke.

A.6.3 Method Return

Bytecode Instruction `return`

Stack [before] ⇒ [after] *... ⇒ [empty]*

Native Code Translation `CALL __DestroyStackFrame`

Description

The `return` bytecode instruction is used to exit the current executing method. The native code translation only requires to create a native `CALL` instruction to the run-time environment's `__DestroyStackFrame` function which is responsible for destroying the method stack frame and returning execution to the caller.

A.6.4 Method Return Instructions

Bytecode Instructions `areturn`, `freturn`, `ireturn`

Stack [before] ⇒ [after] *..., value ⇒ [empty]*

Native Code Translation `CALL __DestroyStackFrame`

Description

The bytecode instructions are used to exit the current executing method and return the value on the stack. The native code translation only requires to create a native `CALL` instruction to the run-time environment's `__DestroyStackFrame` function which is responsible for destroying the method stack frame, returning execution to the caller and also returning the value placed on the stack.

A.7 Exceptions

Bytecode Instruction `athrow`

Stack [before] ⇒ [after] *..., objectref* ⇒ *objectref*

Native Code Translation `CALL __FindACatchHandler`

Description

The `athrow` bytecode instruction is used to throw an exception. The exception object reference will be placed on the stack, therefore the native code translation only requires to create a native `CALL` instruction to the run-time environment function which handles exceptions (the function `__FindACatchHandler`).

A.8 Synchronization Instructions

A.8.1 Gain Ownership of Object

Bytecode Instruction `monitorenter`

Stack [before] ⇒ [after] *..., objectref ⇒ ...*

Native Code Translation
```
CALL __DoMonitorEnter
```

Description

The `monitorenter` bytecode instruction is used to gain ownership of an object, which is used to implement synchronization. The run-time environment function `__DoMonitorEnter` handles the access to gain ownership, and therefore the native code translation is only required to create a native `CALL` instruction to the run-time environment function.

A.8.2 Release Ownership of Object

Bytecode Instruction `monitorexit`

Stack [before] ⇒ [after] *..., objectref ⇒ ...*

Native Code Translation
```
MOV.W @R8, R5
DECD.W R8
DEC.B 0x0002(R5)
```

Description

The `monitorexit` bytecode instruction is used to release ownership of an object, which is used to implement synchronization. The native code translation pops the object reference from the (reference) stack (by copying the value from the reference stack, and decrementing the reference stack pointer). Thereafter, the monitor count is decreased.

References

T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. Envirotrack: towards an environmental computing paradigm for distributed sensor networks. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 582 – 589, 2004.

Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time java compiler. *SIGPLAN Not.*, 33:280–290, May 1998. ISSN 0362-1340.

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10088-6.

Kemal Akkaya and Mohamed Younis. A survey on routing protocols for wireless sensor networks. *Ad Hoc Networks*, 3:325–349, 2005.

I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38:393–422, 2002.

Elvira Albert, Samir Genaim, and Miguel Gomez-Zamalloa. Heap space analysis for java bytecode. In *Proceedings of the 6th international symposium on Memory management*, ISMM '07, pages 105–116, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-893-0.

Rodrigo Allgayer, Marcelo Gtz, and Carlos Pereira. Femtonode: Reconfigurable and customizable architecture for wireless sensor networks. In *Analysis, Architectures and Modelling of Embedded Systems*, volume 310 of *IFIP Advances in Information and Communication Technology*, pages 302–309. Springer Boston, 2009. ISBN 978-3-642-04283-6. 10.1007/978-3-642-04284-328.

Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing jalapeo in java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '99, pages 314–324, New York, NY, USA, 1999. ACM. ISBN 1-58113-238-7.

Denis N. Antonioli and Markus Pilz. Analysis of the java class file format. Technical report, 1998.

A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y. Choi, T. Herman, S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita. A line in the sand: A wireless sensor network for target detection, classification, and tracking. *Computer Networks (Elsevier*, 46:605–634, 2004.

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

Faisal Aslam, Luminous Fennell, Christian Schindelhauer, Peter Thiemann, Gidon Ernst, Elmar Haussmann, Stefan Rührup, and Zastash Uzmi. Optimized Java Binary and Virtual Machine for Tiny Motes. In *Distributed Computing in Sensor Systems*, volume 6131, pages 15–30, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-13650-4.

Faisal Aslam, Christian Schindelhauer, Gidon Ernst, Damian Spyra, Jan Meyer, and Mohannad Zalloom. Introducing takatuka: a java virtualmachine for motes. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, SenSys '08, pages 399–400, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-990-6.

John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003. ISSN 0360-0300.

Árpád Beszédes, Rudolf Ferenc, Tibor Gyimóthy, André Dolenc, and Konsta Karsisto. Survey of code-size reduction methods. *ACM Comput. Surv.*, 35:223–267, September 2003. ISSN 0360-0300.

Bruno Blanchet. Escape analysis for javatm: Theory and practice. *ACM Trans. Program. Lang. Syst.*, 25:713–775, November 2003. ISSN 0164-0925.

Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Querying the physical world. *IEEE Personal Communications*, 7:10–15, 2000.

Per Bothner. Compiling java with gcj. *Linux J.*, 2003:4–, January 2003. ISSN 1075-3583.

Florian Brandner, Tommy Thorn, and Martin Schoeberl. Embedded jit compilation with cacao on yari. In *Proceedings of the 2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, ISORC '09, pages 63–70, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3573-9.

N. Brouwers. A java compatible virtual machine for wireless sensor networks. Master's thesis, Delft University of Technology, 2009.

N. Brouwers, K. Langendoen, and P. Corke. Darjeeling, a feature-rich VM for the resource poor. In *SenSys09*, Berkeley, CA, nov 2009.

Niels Brouwers, Peter Corke, and Koen Langendoen. Darjeeling, a java compatible virtual machine for microcontrollers. In *Companion '08: Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, pages 18–23, New York, NY, USA, 2008a. ACM. ISBN 978-1-60558-369-3.

Niels Brouwers, Peter Corke, and Koen Langendoen. A java compatible virtual machine for wireless sensor nodes. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, SenSys '08, pages 369–370, New York, NY, USA, 2008b. ACM. ISBN 978-1-59593-990-6.

Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The jalapeo dynamic optimizing compiler for java. In *Proceedings of the ACM 1999 conference on Java Grande*, JAVA '99, pages 129–141, New York, NY, USA, 1999. ACM. ISBN 1-58113-161-5.

A. Butters. Total cost of ownership: A comparison of c/c++ and java. Technical report, 2007.

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*, 0:117–125, 2009.

Sigmund Cherem and Radu Rugina. Uniqueness inference for compile-time object deallocation. In *Proceedings of the 6th international symposium on Memory management*, ISMM '07, pages 117–128, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-893-0.

Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. *SIGPLAN Not.*, 34:1–19, October 1999. ISSN 0362-1340.

David Cormie. Jazelle - arm architecture extensions for java applications. Technical report, ARM, 2000.

Alexandre Courbot, Gilles Grimaud, and Jean-Jacques Vandewalle. Efficient off-board deployment and customization of virtual machine-based embedded systems. *ACM Trans. Embed. Comput. Syst.*, 9:21:1–21:53, March 2010. ISSN 1539-9087.

T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling java just in time. *Micro, IEEE*, 17(3):36–43, May/Jun 1997. ISSN 0272-1732.

Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex: an optimizing compiler for object-oriented languages. *SIGPLAN Not.*, 31:83–100, October 1996. ISSN 0362-1340.

Mourad Debbabi, Abdelouahed Gherbi, Lamia Ketari, Chamseddine Talhi, Nadia Tawbi, Hamdi Yahyaoui, and Sami Zhioua. A dynamic compiler for embedded java virtual machines. In *Proceedings of the 3rd international symposium on Principles and practice of programming in Java*, PPPJ '04, pages 100–106. Trinity College Dublin, 2004. ISBN 1-59593-171-6.

Mourad Debbabi, Abdelouahed Gherbi, Azzam Mourad, and Hamdi Yahyaoui. A selective dynamic compiler for embedded java virtual machines targeting arm processors. *Science of Computer Programming*, 59(1-2):38 – 63, 2006. ISSN 0167-6423. ¡ce:title¿Special Issue on Principles and Practices of Programming in Java (PPPJ 2004)¡/ce:title¿ ¡xocs:full-name¿Special Issue on Principles and Practices of Programming in Java (PPPJ 2004)¡/xocs:full-name¿.

Bertrand Delsart, Vania Joloboff, and Eric Paire. Jcod: A lightweight modular compilation technology for embedded java. In *Proceedings of the Second International Conference on Embedded Software*, EMSOFT '02, pages 197–212, London, UK, 2002. Springer-Verlag. ISBN 3-540-44307-X.

I. Demirkol, C. Ersoy, and F. Alagoz. MAC protocols for wireless sensor networks: a survey. *Communications Magazine, IEEE*, 44(4):115–121, April 2006. ISSN 0163-6804.

L. Peter Deutsch and Allan 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*, pages 297–302, New York, NY, USA, 1984. ACM. ISBN 0-89791-125-3.

H. Dubois-Ferriere, R. Meier, L. Fabre, and P. Metrailler. Tinynode: a comprehensive platform for wireless sensor network applications. In *Information Processing in Sensor Networks, 2006. IPSN 2006. The Fifth International Conference on*, pages 358–365, 2006.

Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 15–28, New York, NY, USA, 2006a. ACM. ISBN 1-59593-343-3.

Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2260-2.

Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, SenSys '06, pages 29–42, New York, NY, USA, 2006b. ACM. ISBN 1-59593-343-3.

ECMA International. *Standard ECMA-335 - Common Language Infrastructure (CLI).* 4 edition, June 2006.

Joshua Ellul and Kirk Martinez. Run-time compilation of bytecode in sensor networks. In *Proceedings of the 2010 Fourth International Conference on Sensor Technologies and Applications*, SENSORCOMM '10, pages 133–138, Washington, DC, USA, July 2010a. IEEE Computer Society. ISBN 978-0-7695-4096-2.

Joshua Ellul and Kirk Martinez. Run-time compilation of bytecode in wireless sensor networks. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '10, pages 422–423, New York, NY, USA, April 2010b. ACM. ISBN 978-1-60558-988-6.

Andreas Gal, Christian W. Probst, and Michael Franz. Hotpathvm: an effective jit compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 144–153, New York, NY, USA, 2006. ACM. ISBN 1-59593-332-8.

David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 1–11, New York, NY, USA, 2003. ACM. ISBN 1-58113-662-5.

Fabrice Bellard Charles Consel Gilles Muller, Barbara Moura. Jit vs offline compilers: Limits and benefits of bytecode compilation. Technical report, 1996.

James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1996. ISBN 0201634511.

Samuel Z. Guyer, Kathryn S. McKinley, and Daniel Frampton. Free-me: a static analysis for automatic individual object reclamation. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 364–375, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4.

Ralph Clarke Haygood. Native code compilation in sicstus prolog. In *Proceedings of the eleventh international conference on Logic programming*, pages 190–204, Cambridge, MA, USA, 1994. MIT Press. ISBN 0-262-72022-1.

Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 0321154916.

John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. ISBN 0123704901.

Mohammad Sajjad Hossain, A. B. M. Alim Al Islam, Milind Kulkarni, and Vijay Raghunathan. μsetl: A set based programming abstraction for wireless sensor networks. In *IPSN*, pages 354–365, 2011.

Cheng-Hsueh A. Hsieh, John C. Gyllenhaal, and Wen-mei 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*, pages 90–99, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7641-8.

Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27(5):1–164, May 1992. ISSN 0362-1340.

Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *Proceedings of the ACM 1999 conference on Java Grande*, JAVA '99, pages 119–128, New York, NY, USA, 1999. ACM. ISBN 1-58113-161-5.

Sérgio Akira Ito, Luigi Carro, and Ricardo Pezzuol Jacobi. Making java work for microcontroller applications. *IEEE Des. Test*, 18:100–110, September 2001. ISSN 0740-7475.

Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., New York, NY, USA, 1996. ISBN 0-471-94148-4.

Peter King, Patrick Naughton, Mike DeMoney, Jonni Kanerva, Kathy Walrath, and Scott Hommel. Java code conventions. Sun Microsystems Inc., 1999.

Joel Koshy and Raju Pandey. Vmstar: synthesizing scalable runtime environments for sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 243–254, New York, NY, USA, 2005. ACM. ISBN 1-59593-054-X.

Joel Koshy, Ingwar Wirjawan, Raju Pandey, and Yann Ramin. Balancing computation and communication costs: The case for hybrid execution in sensor networks. *Ad Hoc Netw.*, 6(8):1185–1200, 2008. ISSN 1570-8705.

A. Krall. Efficient javavm just-in-time compilation. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, pages 205–, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8591-3.

Andreas Krall and Reinhard Grafl. Cacao a 64-bit javavm just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, 1997. ISSN 1096-9128.

Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9.

P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. Tinyos: An operating system for sensor networks. In *in Ambient Intelligence*. Springer Verlag, 2004.

Philip Levis and David Culler. Maté: a tiny virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(5):85–95, 2002. ISSN 0163-5980.

Philip Levis, David Gay, and David Culler. Active sensor networks. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 343–356, Berkeley, CA, USA, 2005. USENIX Association.

Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999. ISBN 0201432943.

Tim Lindholm and Frank Yellin. Java card 2.2 off-card verifier. Technical report, 2005.

B. Lo, S. Thiemjarus, R. King, and G. Yang. Body sensor network - a wireless sensor platform for pervasive healthcare monitoring. In *Proceedings of the 3rd International Conference on Pervasive Computing (Pervasive 2005)*, pages 77–80, 2005.

Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, March 2005. ISSN 0362-5915.

Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, New York, NY, USA, 2002. ACM. ISBN 1-58113-589-0.

K. Martinez, J.K. Hart, and R. Ong. Environmental sensor networks. *Computer*, 37(8): 50–56, Aug. 2004. ISSN 0018-9162.

John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960. ISSN 0001-0782.

Harlan McGhan and Mike O'Connor. Picojava: A direct execution engine for java bytecode. *Computer*, 31:22–30, October 1998. ISSN 0018-9162.

Jim S. Miller and Susann Ragsdale. *The Common Language Infrastructure Annotated Standard*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 0321154932.

James George Mitchell. *The design and construction of flexible and efficient interactive programming systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1970. AAI7104538.

Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks with logical neighborhoods. In *Proceedings of the first international conference on Integrated internet ad hoc and sensor networks*, InterSense '06, New York, NY, USA, 2006. ACM. ISBN 1-59593-427-8.

Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. Harissa: a flexible and efficient java environment mixing bytecode and compiled code. In *Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 3*, pages 1–1, Berkeley, CA, USA, 1997. USENIX Association.

Ryan Newton and Matt Welsh. Region streams: functional macroprogramming for sensor networks. In *Proceeedings of the 1st international workshop on Data management for sensor networks: in conjunction with VLDB 2004*, DMSN '04, pages 78–87, New York, NY, USA, 2004. ACM.

Doug Palmer. A virtual machine generator for heterogeneous smart spaces. In *VM'04: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*, pages 1–1, Berkeley, CA, USA, 2004. USENIX Association.

Raju Pandey and Joel 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*, page 11, New York, NY, USA, 2006. ACM. ISBN 1-59593-427-8.

Kris Pister. On the limits and applications of mems sensor networks. Technical report, UC Berkeley, 2001.

Michael P. Plezbert and Ron K. Cytron. Does "just in time" = "better late than never"? In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 120–131, New York, NY, USA, 1997. ACM. ISBN 0-89791-853-3.

Joseph Polastre, Robert Szewczyk, and David Culler. Telos: enabling ultra-low power wireless research. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 48, Piscataway, NJ, USA, 2005. IEEE Press. ISBN 0-7803-9202-7.

Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for applications: A way ahead of time (wat) compiler. Technical report, Tucson, AZ, USA, 1997.

Ian A. Rogers. *Optimising Java Programs Through Basic Block Dynamic Compilation.* PhD thesis, University of Manchester, 2002.

Sensirion. Sht11 datasheet, 2010.

Alec Sharp. *Smalltalk by Example: The Developer's Guide.* McGraw-Hill, Inc., New York, NY, USA, 1st edition, 1996. ISBN 0079130364.

Nik Shaylor. A just-in-time compiler for memory-constrained low-power devices. In *Proceedings of the 2nd Java™ Virtual Machine Research and Technology Symposium*, pages 119–126, Berkeley, CA, USA, 2002. USENIX Association. ISBN 1-931971-01-3.

Nik Shaylor, Douglas N. Simon, and William R. Bush. A java virtual machine architecture for very small devices. *SIGPLAN Not.*, 38:34–41, June 2003. ISSN 0362-1340.

Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual machine showdown: Stack versus registers. *ACM Trans. Archit. Code Optim.*, 4(4):1–36, 2008. ISSN 1544-3566.

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts.* John Wiley & Sons, Inc., New York, NY, USA, 6th edition, 2001. ISBN 0471417432.

Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. Java on the bare metal of wireless sensor devices: the squawk java virtual machine. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 78–88, New York, NY, USA, 2006. ACM. ISBN 1-59593-332-6.

Chavalit Srisathapornphat, Chaiporn Jaikaeo, and Chien-Chung Shen. Sensor information networking architecture. *Parallel Processing Workshops, International Conference on*, 0:23, 2000. ISSN 1530-2016.

L. Steinfeld and L. Carro. The Case for Interpreted Languages in Sensor Networks. In *Analysis, Architectures and Modelling of Embedded Systems*, pages 279–+, 2009.

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.*, 39(1):175–193, 2000. ISSN 0018-8670.

Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a java just-in-time compiler. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 180–195, New York, NY, USA, 2001. ACM. ISBN 1-58113-335-9.

Ryo Sugihara and Rajesh K. Gupta. Programming models for sensor networks: A survey. *ACM Trans. Sen. Netw.*, 4(2):8:1–8:29, April 2008. ISSN 1550-4859.

Sun Microsystems. Java platform debugger architecture - java debug wire protocol. `http://java.sun.com/products/jpda/doc/jdwp-spec.html`.

Texas Instruments. Msp430x1xx family user's guide (rev. f), 2004. `http://focus.ti.com/lit/ug/slau049f/slau049f.pdf`.

Texas Instruments. Msp430f15x, msp430f16x, msp430f161x mixed signal microcontroller, 2009. `http://www.ti.com/lit/gpn/msp430f1611`.

Texas Instruments. Cc2420 datasheet, 2010.

Isabella Thomm, Michael Stilkerich, Christian Wawersich, and Wolfgang Schröder-Preikschat. Keso: an open-source multi-jvm for deeply embedded systems. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 109–119, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0122-0.

Paul Tyma. Why are we using java again? *Commun. ACM*, 41(6):38–42, 1998. ISSN 0001-0782.

David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *SIGSOFT Softw. Eng. Notes*, 9:157–167, April 1984. ISSN 0163-5948.

Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: scalable threads for internet services. *SIGOPS Oper. Syst. Rev.*, 37:268–281, October 2003. ISSN 0163-5980.

Brett Warneke, Kristofer S. J, and Smart Dust. Smart dust: Communicating with a cubic-millimeter computer. *Classical Papers on Computational Logic*, 1:372–383, 2001.

Mark Weiser. The computer for the 21st century. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3:3–11, July 1999. ISSN 1559-1662.

Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, MobiSys '04, pages 99–110, New York, NY, USA, 2004. ACM. ISBN 1-58113-793-1.

Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioglu, and Erik Altman. Latte: A java vm just-in-time compiler with fast and efficient register allocation. *Parallel Architectures and Compilation Techniques, International Conference on*, 0:128, 1999. ISSN 1089-795X.