

Balancing Computation and Code Distribution Costs: The Case for Hybrid Execution in Sensor Networks

Ingwar Wirjawan, Joel Koshy, Raju Pandey, Yann Ramin

Department of Computer Science

University of California, Davis

{wirjawan|koshy|pandey|ramin}@cs.ucdavis.edu

ABSTRACT

Virtual machines (VM) are promising as system software in networks of embedded systems and pervasive computing spaces. VMs facilitate the development of platform-independent applications with small footprints to enable low cost application distribution and evolution. A major impediment to their more widespread acceptance is the performance overhead of interpretation. Compiling VM bytecode to native instructions addresses this issue, but can increase footprint and code distribution costs. Thus, there is an important tradeoff between cost of computing, and cost of communication due to code distribution. In this paper, we describe a remote Just-In-Time (JIT) compilation service for the VM[★] framework that is effective in combining interpretation with native execution to arrive at an efficient hybrid execution configuration. The principles apply to any VM or middleware used to develop applications in sensor networks.

1. INTRODUCTION

Real world deployments of wireless sensor networks (WSN) are increasingly hierarchical in their configuration. These networks tend to have multiple tiers with sensing at the edges, in-network processing and aggregation at intermediate tiers, and relaying and cluster management at the highest tiers. Diversity is also present within individual tiers. This growing trend toward heterogeneity in WSNs is a compelling reason to use virtual machine (VM) abstractions for deploying applications.

There are several benefits in using VMs [13, 14, 12] in this domain. First, VMs allow applications to be developed uniformly across the various platforms. Rather than having to tailor applications to the limitations of the least common denominator of all devices, platform-independent applications can be written using VM abstractions, and the VM implementation is scaled to meet application requirements within the resource constraints of the devices. Second, VMs provide a clean separation of applications system software which reduces the cost of reprogramming applications after deployment, since the system software usually does not need to change. Finally, VMs mask the idiosyncratic variations among the devices through a common execution framework. The common intermediate representation becomes the basis for application distribution, management and interoperability.

The primary argument against using VMs is that they introduce a layer of indirection in application execution. Thus, VM programs execute slower, and consume more energy than their native counterparts. For example, the Maté VM [13] reports a 33.5:1 overhead for a simple `and` operation, and VM[★] [12] has a 5:1 overhead for an integer addition (`iadd`). (VM[★]'s overhead is comparatively less because it is currently single-threaded and uses a *threaded* interpreter). The overheads are typically due to the stack-oriented nature of these VMs, bytecode interpretation overheads, and overheads peculiar to the VM specifications such as type checks or synchronization sequences. The scheduling policies of the supporting operating system (OS) also have effects on VM efficiency. What is needed is an execution environment that provides the VM benefits of platform-independent application development, high level of abstraction, and network reprogrammability, without incurring unacceptable overheads.

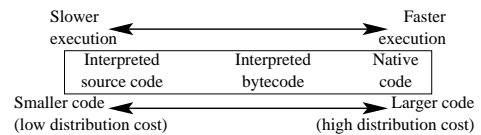


Figure 1: Interpretation and native execution time-space tradeoff [3].

In this paper, we argue that a hybrid execution environment that enables co-execution of platform-independent VM instructions with native instructions answers this need, and is effective in addressing the problems associated with pure native or pure virtual execution environments.

In the interpreted mode, platform-independent bytecode is executed by an interpretive execution engine. In the native mode, a light-weight native interface is used to access natively implemented functionality. This computing model is important, because it allows identifying a balance between two key costs in WSN applications: communication (code distribution), and computation. Interpreted code is platform-independent and can be distributed at lower cost due to more compact bytecode. However, it is less efficient due to interpretation overheads. Native code is efficient, but is more expensive to distribute and is platform-dependent. Thus, there is a fundamental tradeoff between execution efficiency and cost of code distribution (Figure 1). By distribut-

ing platform-independent bytecode at low cost and implementing compute-intensive operations in native code, a very efficient overall configuration can be achieved. Another motivation for using a hybrid execution model is that VM application binaries are more compact than native code. Some complex algorithms can have extremely high footprints and will not even fit within the program memory of some platforms. Not only is it easier to implement complex algorithms using high level VM abstractions (e.g., Java libraries), but VM abstractions and specializing compute-intensive methods using native code helps achieve reasonable performance within the memory constraints of the platform.

In most VMs, the task of translating bytecode to native code is handled by an integrated Just-In-Time (JIT) compiler. Through online profiling and JIT compilation, the hybrid execution capability of the VM can be exploited to arrive at an efficient execution configuration at runtime. Implementing a realistic hybrid execution environment in WSNs is challenging for three reasons.

Resources: WSN nodes have limited resources and are incapable of compiling bytecode to native code locally. JIT compilers are non-trivial programs that have significant memory footprint. The compilation process itself is compute-intensive, resulting in energy drain and latencies that are unacceptable in most applications.

Profiling overheads: With a JIT compilation service, it is necessary to perform execution profiling, because proper selection of candidate methods for compilation is critical to its effectiveness. Static selection schemes are straightforward to implement but may not be capable of capturing the range of execution behavior exhibited after a deployment is in operation. For this reason, energy-efficient dynamic profiling is an important component of the system. In designing a profiling mechanism, the runtime overhead of intercepting method invocations and the distributed costs of transmitting profile information to the compilation server need to be reduced. The challenge is to ensure low CPU overheads at the nodes, and limit the size and number of messages used in profile collection.

Native code generation and linking: The hybrid execution model creates a size-efficiency tradeoff. To obtain maximum benefit, the compilation service should be capable of operating with at least method granularity, and the native code footprint should be reduced as much as possible. Thus, the intermediate representations used to generate native code need to be amenable to aggressive compiler optimizations. Also, in order to co-exist with interpreted code efficient primitives for native-virtual interaction should be in place. Finally, the framework should not be tightly coupled to any particular architecture, since real deployments that benefit from VM approaches contain multiple platforms.

A viable alternative to local JIT compilation is to use a proxy compilation service. In this approach, candidate methods for compilation are determined and their identifying information (e.g., bytecode or method names) are sent to a more capable node dedicated for the purpose of cross-compiling bytecode for the node. This saves resources on the target device by offloading the expensive compilation

and optimization tasks to the more powerful compilation server. Although sending the request and downloading native code compiled from bytecode is energy-intensive, the cost is amortized over the lifetime of the application by carefully selecting only frequently invoked methods for compilation.

We have implemented a prototype remote JIT compilation service for VM[★] that includes a remote bytecode to native translation tool and a distributed profiling system. By offloading compilation to an external resource, JIT compilation is achievable even in the resource-constrained WSN domain. Our translation tool is built over the Soot Java optimization framework [29], and currently translates Java bytecode to GNU C which is then compiled to native code that can be linked with the VM. By generating GNU C, the tool is not bound to any particular platform, and we can compile Java methods to native code on platforms that have backends implemented by the GNU compiler collection. In fact, we have successfully used the framework on both the Mica2 [9] and Telos [22] sensor node platforms. Our test applications yield speedups of up to 77% in application configurations suitable for JIT compilation, with program footprints increasing by a factor of 5.58 for Mica2 and 3.74 for Telos. Overall stack requirements increase depending on application call patterns. The results show that through selective compilation of performance critical methods, the efficiency of VM-based execution environments can be significantly improved, making them viable as a basis for software development, deployment and execution in WSNs.

The remainder of the paper is organized as follows. In Section 2, we discuss the main overheads of virtual execution environments that JIT compilation addresses. We present a conceptual overview of the JIT compilation service as a means to realize the hybrid execution model in Section 3. In Section 4 we present an analytical model that provides some guidelines for determining a suitable native-virtual configuration. Section 5 gives details of our current implementation, along with brief descriptions of supporting infrastructure such as VM[★] and Soot. In Section 6, we evaluate the performance of the system focusing on the speedups and memory overheads. Section 7 contains a summary of related work in JIT compilers and other VM optimizations. Section 8 contains concluding remarks and future directions.

2. VIRTUAL MACHINE OVERHEADS AND JIT COMPILATION

We begin this discussion by identifying the main costs in the Java Virtual Machine's (JVM) interpretive execution model. The additional cost of running a Java program on any platform arises from two factors: (i) programming language abstractions and (ii) interpretive mode of execution. Programming language abstractions that support dynamic name bindings, higher level abstractions, and dynamic runtime management (such as inheritance, threading and exception handling, heap-allocated and managed objects and arrays and runtime reflection) require runtime resolution and resource management that add substantial runtime cost. In this paper, we do not focus on the language-specific costs. The interpreter overhead consists of (i) fetching the next bytecode, (ii) decoding and starting the instruction, and (iii) executing the bytecode implementation. There are ad-

ditional sources of overhead in interpretation and the code snippet shown in Figure 2 illustrates several of these. When the boolean member of the object is true, a character variable is incremented and passed as an argument in a virtual method invocation.

Java source	Compiled bytecode
if (o.stale) {	aload_2
c += 1000;	getfield #cp _i
o.resetField(c)	ifeq @lb11
...	iload_1
	sipush 1000
	iadd
	i2c
	istore_1
	aload_2
	iload_1
	invokevirtual #cp _{i+1}
	lb11: ...

Figure 2: Interpreter and stack overheads.

We can classify the costs into four main categories:

Interpretation overhead: Bytecodes need to be fetched by incrementing and performing an indirect read of the instruction pointer. This is followed by the decode step before dispatching for actual execution of the bytecode. The fetch and decode steps do not contribute to effective CPU utilization.

Stack overhead: These overheads are strictly VM-specific as some VMs may be register-oriented. The execution of the stack-oriented bytecode can be inefficient in itself, given that it is implemented over typically register-oriented hardware. Operands to the *iadd* instruction for instance need to be pushed onto the stack. The *iadd* bytecode implementation needs to pop them, add them, and push the result back onto the stack. A register-oriented architecture can potentially halve the number of cycles required to do the entire operation.

Redundant operations: Due to the stack-oriented computing model, data needs to be reloaded onto the stack for each access. In Figure 2, the two accesses to *o* results in two separate stack push operations. A register-oriented architecture avoids this by binding data to registers within a scope.

VM-specific overheads: Some overheads are peculiar to individual VM architectures. For example, the JVM requires stack entries to be four bytes wide. Thus, even the two byte character variable and the constant 1000 in Figure 2 are promoted to integers when pushed onto the stack. When storing the result, it is cast back to a character. Complex object models, garbage collection, synchronization procedures, runtime type checking and other security checks mandated by some VM specifications are some other examples of VM-specific overhead.

While these do not cover all the overheads of VM execution models, they are representative of the most typical costs across VM architectures. Various techniques have been used to address these costs. Threaded code [4], inlined threading [20], and superoperators [23] reduce interpreter loop overheads. Stack caching [6] and stack folding oper-

ations [28] reduce stack overheads. Stack caching also addresses the stack reloading problem to some degree. JIT compiling techniques [3] reduce or eliminate *all* of these overheads.

Classic JIT compilation involves translation from VM bytecode to native machine code at runtime. Bytecodes are analyzed (usually at the level of basic blocks) and transformed to an intermediate representation (IR) which is more amenable to various compiler optimizations. The IR is typically generated by "simulating" the bytecode sequence to determine a suitable register allocation for the basic block. The JIT compiler maintains its own stack which mirrors the runtime status of the operand stack. The simulated stack's entries contain information about intermediate results of the computation. Stack entries for values that are not yet produced record the operations that need to be performed. Entries for computed values indicate the register or memory location that holds the value. The resulting effect is that basic blocks of stack-oriented bytecode are translated to register-oriented code.

3. REMOTE JIT COMPILATION SERVICE

In this section, we describe the overall remote JIT compilation service architecture that supports a hybrid execution environment in WSNs. The JIT compilation service can be used in a continuous profile-JIT compile-redeploy cycle as shown in Figure 3. There are four distinct phases in this architecture: (i) Deployment of application and device-specific virtual machine environment (ii) Instrumentation (iii) Compilation (iv) Remote linking. We describe each below.

The **development and deployment phase** involves building and deploying an execution environment for an application. In this phase, programmers use a high level language to develop VM applications. We assume a VM-specific execution format exists (Java bytecode in our case). Configuration and synthesis tools [31] analyze the application binary, determine the set of VM runtime components needed to run the application, and generate a resource-efficient runtime environment. The generated VM also includes instrumentation code that intercepts method calls and collects profiling data. A code dissemination tool distributes the VM binary along with the application to the sensor nodes.

After the application and runtime environment are deployed, the **instrumentation phase** is entered. In addition to the tasks of orchestrating the build process of the runtime environment, the base station is also responsible for maintaining state about the deployment. This includes application classes, deployed program memory images and other meta information necessary for incremental updates to the deployment. It is possible to reduce this requirement by increasing the semantic information present at the nodes. In our framework, we have stripped away most of this information to reduce node memory footprint and energy required to transmit profile data. Also, we have adopted the policy of pushing as much of the administrative overhead to the base station. Node collect online profiling data and send it to the base station. There are several strategies that nodes can use for profiling. These are discussed further in Section 5.2. Based on data collected from the network, the base station makes the final decision as to which methods will be JIT

compiled.

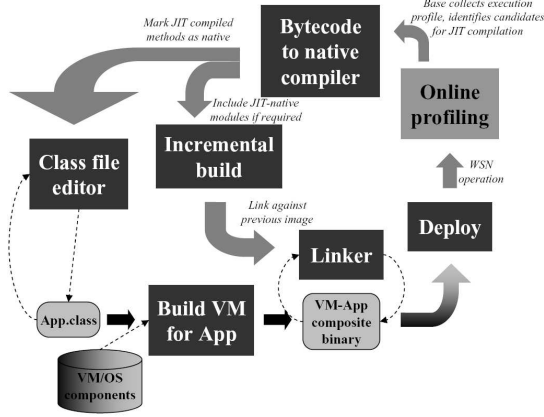


Figure 3: JIT compilation service cycle.

In the **compilation phase**, the JIT compiler at the base station translates performance critical method bytecode to native code. The precise steps needed to do this can vary. Optimized machine code may be generated directly from bytecode or via traditional intermediate code as long as the intermediate representation is amenable to aggressive optimization in the final stages of compilation. Depending on the kind of intermediate code used, different compiler optimizations may be more effective. A critical challenge for traditional JIT compilers is to generate efficient native code without incurring the large costs of traditional optimization techniques. This is not only to reduce the resources required for its operation but also to reduce the latency involved in JIT compilation. In traditional systems, code size is not a critical issue. However, in the WSN domain, since code size has direct impact on code distribution cost and memory requirements, the JIT compiler must find a balance between size and execution efficiency. Using a remote JIT compilation service allows us to use aggressive optimizations at the base station where CPU resources are not a major concern. At the same time, the base station controls the compilation process to optimize for size as well. In Section 5.3, we describe our JIT compiler framework and its implementation.

In the **incremental linking phase**, the base station incrementally links the generated native binary against the existing VM. This is necessary, as there may be references from the code to VM routines (mainly JIT library routines required to interact with the VM data space). There will be no direct calls to the JIT compiled method from the VM. The method will be invoked indirectly by the application class, through the VM’s native interface. Thus, the object is essentially position-independent, which simplifies linking considerably. This also means that the code can be distributed without any relocation process, and the node can store it in internal flash at any address specified by the base station.

4. ANALYTICAL GUIDELINES

Applications can execute in three modes: interpretation, native, and hybrid. Interpreted code is compact with low distribution overheads, but costly to execute. Native code is an order of magnitude larger, but runs much faster. The hybrid

mode tries to balance the overall cost by trying to reduce the cost of distribution and the cost of interpretation. In this section, we develop an analytical framework that provides hints for arriving at an efficient native-VM boundary. The following table defines the different parameters used in the analysis.

j	% of the application candidate for JIT compilation
k	s_{an}/s_{ab} , denoting average code bloat from byte code to a native binary
m	Number of times non-candidate portion is executed
n	Number of times candidate portion is executed
r	Number of times runtime is executed
s_{ab}	Size (in bytes) of application (byte code)
s_{an}	Size (in bytes) of application (binary)
s_{f_b}	Size (in bytes) of non-native method f
s_{f_n}	Size (in bytes) of native method f
s_r	Size (in bytes) of language runtime and libraries
t_d	Average distribution cost per byte per node
t_i	Average interpretive execution cost per byte
t_n	Average native execution cost per byte

The quantity j denotes the fraction of the application (called the *candidate* portion) that when implemented natively will potentially benefit the application. With the conclusions made below, given a set of candidate methods we can estimate the criteria under which JIT compiling those methods will be beneficial. A reasonable simplifying assumption that we make is that j is the same for both native and interpreted code. That is, if a method f is a candidate, j in interpreted mode is s_{f_b}/s_{ab} , and is equal to the corresponding j in native mode, which is s_{f_n}/s_{an} .

Interpretation mode

The distribution cost (D_i) in the interpretation mode includes the costs of sending both the runtime environment binary and the application:

$$D_i = s_r t_d + s_{ab} t_d$$

The execution cost (E_i) involves the runtime environment overhead, and cost of interpreting the candidate and non-candidate portions:

$$E_i = r s_r t_n + m(1-j)s_{ab} t_i + n j s_{ab} t_i$$

The total cost of execution (C_i) is $D_i + E_i$. Interpretation requires $s_r + s_{ab}$ bytes of program space.

Native mode

The distribution cost (D_n) in the native mode includes the costs of sending both application and runtime environment native binaries:

$$D_n = s_r t_d + s_{an} t_d$$

The execution cost (E_n) involves the runtime environment overhead and cost of executing the candidate and non-candidate portions natively:

$$\begin{aligned} E_n &= r s_r t_n + m(1-j)s_{an} t_n + n j s_{an} t_n \\ &= r s_r t_n + m k (1-j)s_{ab} t_n + n k j s_{ab} t_n \end{aligned}$$

In the above, we have replaced s_{an} by $k s_{ab}$, which highlights the code bloat. The total cost of execution (C_n) is $D_n + E_n$. The native execution mode requires $s_r + s_{an}$ bytes of program space.

Hybrid mode

In the hybrid mode, the distribution cost (D_h) also includes the cost of sending the JIT-compiled code:

$$D_h = s_r t_d + s_{ab} t_d + k j s_{ab} t_d$$

The execution cost (E_h) involves the runtime environment cost, interpreting the non-candidate portion, and natively executing the candidate portion:

$$\begin{aligned} E_h &= r s_r t_n + m(1-j)s_{ab} t_i + n j s_{ab} t_n \\ &= r s_r t_n + m(1-j)s_{ab} t_i + n k j s_{ab} t_n \end{aligned}$$

The total cost of execution (C_h) is $D_h + E_h$. The hybrid execution mode requires $s_r + s_{ab} + j k s_{ab}$ bytes of program space.

Cost-benefit implications

The overhead of interpreting bytecode with respect to the hybrid mode is:

$$\begin{aligned} C_i - C_h &= n j s_{ab} t_i - k j s_{ab} t_d - n k j s_{ab} t_n \\ &= j s_{ab} (n t_i - k t_d - n k t_n) \\ &> 0 \text{ if} \\ n(t_i - k t_n) - k t_d &> 0 \\ n &> \frac{k t_d}{t_i - k t_n} \\ n &> \frac{k(t_d/t_n)}{(t_i/t_n - k)} \end{aligned}$$

The above inequality suggests that if n (number of times candidate portion is executed) is larger than $k(t_d/t_n)/(t_i/t_n - k)$, interpretation will have larger execution overhead, despite the hybrid mode's extra cost of sending the JIT compiled binary ($j s_{ab}$ bytes). The inequality also highlights the tradeoffs between the distribution and execution costs. When t_i/t_n (interpretation slow down) is less, n needs to be higher, highlighting the dominance of distribution cost: in hybrid execution, the candidate portion of the code will need to be executed more often to offset the cost of sending the JIT compiled binary. Further, larger the value of t_d/t_n (distribution overhead), larger the value of n , highlighting the fact that with higher cost of distribution, the candidate portion needs to be executed more often for any benefit to be observed. n is directly related to k , the code bloat factor. For larger k , the cost of distributing the binary is higher, which necessitates a larger n . For example, if $k = 5, t_d/t_n = 20, t_i/t_n = 70, n > 100/65$. Thus, for $n \geq 2$, the hybrid mode will outperform the interpretation mode. Given the reactive nature of most WSN applications, the above analysis shows that executing the performance critical code in the native mode will speed up the overall execution of the application. The hybrid mode requires an additional $j k s_{ab}$ bytes of storage, motivating a fairly low value of j .

The overhead of running the program in the hybrid mode

with respect to native execution is:

$$\begin{aligned} C_h - C_n &= s_{ab} t_d + k j s_{ab} t_d + m(1-j)s_{ab} t_i \\ &\quad - k s_{ab} t_d - m k (1-j)s_{ab} t_n \\ &= s_{ab} t_d - k(1-j)s_{ab} t_d + (1-j)s_{ab}(m t_i - m k t_n) \\ &= s_{ab} t_d (1 - k(1-j)) + (1-j)m s_{ab}(t_i - k t_n) \\ &> 0 \text{ if} \\ m &> \frac{t_d(k(1-j) - 1)}{(1-j)(t_i - k t_n)} \\ m &> \frac{(t_d/t_n)(k(1-j) - 1)}{(1-j)(t_i/t_n - k)} \end{aligned}$$

The above inequality suggests that as long as m (number of times the non-candidate code is executed) is greater than $t_d/t_n(k(1-j) - 1)/(1-j)(t_i/t_n - k)$, hybrid execution is more costly. When t_d/t_n is higher, m needs to be larger, highlighting the effect of code distribution on native execution. Similarly, when t_i/t_n is larger, m needs to be smaller, highlighting the cost of interpretation. When m is less than $t_d/t_n(k(1-j) - 1)/(1-j)(t_i/t_n - k)$, the hybrid mode outperforms native execution. This represents the case when the cost of distributing extra binary in the native mode dominates the cost of interpretation in the hybrid mode. The native mode requires an additional $k s_{ab} - s_{ab} - j k s_{ab}$ bytes compared to the native mode. For small j , the difference in size may be fairly large.

5. IMPLEMENTATION

We have implemented the system for the Mica2 and Telos platforms. Mica2 nodes contain the Atmel ATmega128 microcontroller [9] with 128K program memory, and 4K internal SRAM. Communication is through a Chipcon CC1000 transceiver. They may be interfaced to external sensor boards, and include 512K of external flash memory. The TelosB platform [22] is equipped with a Texas Instruments MSP430F1611 microcontroller — a 16-bit von Neumann processor with 48KB of flash-based program memory and 10KB of SRAM.

We begin with an overview of the supporting VM[★] infrastructure, focusing on its salient features that are relevant to the compilation service. We then describe the infrastructure needed for collecting profiling data from distributed sensor nodes in Section 5.2. The VM[★] JIT compilation framework uses Soot [29] for compiling performance critical methods. Section 5.3 provides a brief overview of Soot and describes the VM[★] JIT compiler. The base station uses a remote linker to integrate generated binary on sensor nodes. Section 5.4 briefly describes the overall process.

5.1 VM[★]

VM[★] is a framework containing a lightweight JVM implementation. It forms the basis for building resource-efficient VMs specialized for application-platform pairs. It is based on a virtual execution environment built using a fine-grained software synthesis tool [31] and uses incremental linking [11] to allow efficient incremental updates after deployment. The application developer writes a high level application in Java and compiles it to bytecode. The synthesis tool analyzes the bytecode to determine the required VM and OS components, and generates a runtime environment precisely tailored for the application-platform combination. The tasks of application class preparation, VM synthesis, and VM updates are

orchestrated by the base station’s registry, which maintains state information for the deployment.

Java classes typically have a large footprint and cannot be stored on the WSN nodes. VM[★] uses a compacted class file format to deal with this. As a result, the bytecode formats are slightly different from standard bytecode. The main strategy used in compaction is to eliminate redundant string information and encode references as integers. (Standard bytecode uses strings containing fully qualified names for references). While we lose some flexibility in doing so, the savings in footprint are substantial. Moreover, using these encodings reduces the cost of distributed profiling used by the JIT compilation service since lengthy string references do not need to be sent.

VM[★]’s execution engine provides the option of using both threaded and traditional interpretation. Application classes can be stored either in flash or SRAM. Storing them in SRAM yields better performance. For most applications however, the class footprints are too large for SRAM and classes are “ROMized” by default. Platform-specific functionality is implemented through a light-weight native interface. Note that this is not equivalent to using platform or domain-specific bytecodes which reduces application portability. Rather, the VM uses a general purpose instruction set and switches to native mode for binary execution. Native methods in VM[★] bytecode will have their code length fields set to zero. The method invocation bytecode implementations check this field and determine whether switch to native mode is necessary. The switch between native and virtual execution usually involves transferring data between the virtual stack and the native stack. Arguments to native methods are fetched from the Java stack and return values if any are explicitly copied onto the Java stack. The VM native interface provides several functions for exchanging data between the interpreter and native execution spaces. Although native execution is much faster than interpretation, the switch can be expensive. Thus, excepting some platform-specific short computations that *need* to be implemented using the native interface, native code is most effective when implementing compute-intensive functionality.

5.2 Distributed method profiling

Most VM implementations use some form of dynamic profiling of the interpreted code to identify *hot methods*. Examples include invocation counts, duration of execution (of methods), method size, call back edges, etc. JIT compilers exploit both the program structure and runtime execution profiles to make fairly accurate predictions on the benefit of compiling specific methods. Details of the different schemes can be found in [26, 2]. In the WSN domain, the selection of a small set of performance critical methods is equally important. In fact, the well known 80-20 principle should apply to an even greater degree in this domain, because WSN applications are generally small and characterized by repeating compute-sleep patterns, with the cycle interrupted by occasional runtime service related computations.

The VM[★] JIT compilation framework implements a distributed infrastructure for identifying a small set of performance critical Java methods. Nodes collect profiling data

about performance critical methods, and send the data over the radio to the base station. The design of the infrastructure is driven by several factors. First, sensor nodes cannot store large amounts of profiling data. Second, nodes cannot perform compute intensive algorithms for profiling as they have significant energy costs. Third, since VM[★] class files are compacted Java classes with limited meta-information, nodes cannot use the program structure information for profiling. Lastly, the overhead of sending the data over the radio should be minimized. This limits the nature and size of runtime profiling data that can be collected.

There are three key components of the infrastructure: data collection, Data distribution and aggregation, and profile scheduling. We describe each in turn below.

Data collection: Each node runs a software stack consisting of an operating system (OS[★]) which interfaces with the sensor hardware and manages system resources (memory, concurrency, radio, etc), and a virtual machine (VM[★]). We have modified the Java bytecodes that handle method invocation (`invokestatic`, `invokespecial` and `invokevirtual`) to record invocation frequency. The runtime environment maintains a small cache that stores tuples of the form `<class_id, method_id, count>`, where `class_id` and `method_id` correspond to the encodings assigned to classes and methods during the class file rewriting stage described in Section 5.1, and `count` is the invocation count. The current implementation maintains information for six methods, which requires about 25 bytes. This is small enough to be stored in a single message packet. On systems with larger SRAM (such as the Telos and XYZ platforms), the profile data cache may be scaled to store more profiling data. The cache also includes a lock, so that cache accesses by the runtime and data distribution threads are synchronized. The VM[★] runtime environment collects data until they reach a certain threshold.

Profile data distribution and aggregation: Once nodes have collected runtime profile data, they transmit the data to the base station for analysis. Given the multi-hop nature of the network, the data must be routed through the intermediate nodes. The JIT compilation infrastructure contains a routing and aggregation protocol that reduces the number and size of the messages in the network by using a greedy approach to aggregate profiling data at the intermediate nodes. The routing and aggregation protocol are currently based on the shortest distance routing protocol. There are two phases in the protocol.

In the first phase, the protocol builds a routing tree for distributing data, and each node estimates its shortest hop distance from the base station. Nodes initially set their distance estimates to a large number. After the base station advertises its distance as zero, nodes periodically advertise their distance estimates from the base station. Nodes update their estimates if they receive a packet from a neighbor that provides a shorter path to the base station. The protocol can terminate when nodes have stabilized, or continue in the background to deal with node failures.

In the second phase, nodes use the routing tree to send profile data to the base station. A node packs the profile information and its distance estimate into a single message

packet, and broadcasts it. A node that receives a packet from a node that is farther away from the base station, aggregates the received profile data with its local cache. Our current aggregation scheme is very simple. It selects the six most frequently called methods. After a small delay, the node forwards the data to other nodes, which repeat this process until the data reaches the base station. The base station then uses the profile data aggregated from all nodes to select candidate methods for compilation.

Profile scheduling scheme: Given that the WSN domain contains autonomous nodes which run similar code, the issue here is: should all nodes be instrumented? If not, which nodes should be selected for profiling and how does this selection take place? After selecting profile nodes, should all of them send their caches to the base station? In the simplest case, all nodes collect profiling data and send it to the base station. The problem with this approach is that many nodes will waste energy by producing redundant data and sending many messages into the network, in addition to incurring the instrumentation costs. Another approach is to use a node close to the base station to do the profiling, and send the data directly to the base station. This approach is not reliable as the node may fail. Also, it is unfair to the node and will cause it to deplete its energy faster. Moreover, the execution profile from that node may not be a good representative of the deployment as a whole. We are currently exploring several scheduling schemes that try to balance low cost of profiling and messaging with reliability and fairness requirements. In one approach, the base station constructs a static scheduling profile for each node. In this profile, it assigns profiling of specific methods to certain nodes. The nodes only collect data about these methods and send them to the base station. The second approach is to allow nodes to instrument method invocations, but randomly select themselves for sending data. This reduces the number of messages in the network. A third possibility is to have each node flip a coin before instrumenting. If selected, they profile and send data. Thus, on the average, $1/n$ of the nodes perform profiling, and each node sends about $1/n$ of the total messages. Our future work will analyze the effectiveness and performance characteristics of the different scheduling schemes.

5.3 Compiling bytecode to native code

Soot [29] is a Java bytecode analysis and optimization framework. It essentially converts stack-oriented bytecode to register-oriented code suitable for optimizations, and converts the optimized register code to optimized stack-oriented bytecode. Soot provides three intermediate representations: *BAF*, *Jimple*, and *Grimp*. **BAF** is a stack-based intermediate representation of Java bytecode. BAF supports typed instructions, fully resolved constant pool references, and explicitly named local variables, which make bytecode analysis simpler. **Jimple** is a compact three-address bytecode representation that is much easier to optimize than stack-oriented code. Jimple instructions are typed and stackless, and are ideal for performing both traditional optimizations and advanced optimizations applicable to Java such as virtual method call resolution. It is also convenient for translation to procedural languages. The Java bytecode to C tool that we developed (*j2c*) uses Jimple as a backend. **Grimp** is an unstructured representation of Java bytecode. It con-

tains features that make it better for translation to bytecode than Jimple.

These intermediate representations can be used for their individual features, but they are closely coupled to each other within the Soot optimization framework. A typical bytecode optimization pipeline consists of converting Java bytecode to BAF, which is then converted to Jimple, which in turn is converted to Grimp. Grimp is then converted to BAF, and BAF to Java bytecode. Stage-specific optimizations are applied during the process.

The transformation pipeline we used to implement the *j2c* tool is shown in Figure 4. Java sources are not required, as the tool operates on bytecode directly. The application class is registered at the base station, which compacts the class and generates an application dictionary `app.d` which contains mappings from fully qualified names to `class_ids` and `method_ids`. The class is also fed to the Soot pipeline, which transforms it to BAF to perform preliminary bytecode optimizations and analyses. This produces three-address Jimple code which goes through various Jimple optimizations. The *j2c* compiler transforms the Jimple code into C. Statements and expressions that require access to the VM data space and services use a special JIT API. The JIT API contains 42 JIT library routines used for interaction between native code and the VM. Table 1 contains a list of some of these routines.

	Description
<code>jit_get_field_four_bytes</code>	Access four byte instance field.
<code>jit_invokevirtual</code>	Invoke a virtual method.
<code>jit_invokespecial</code>	Invoke a private virtual method, or a constructor.
<code>jit_pop_two_entries</code>	Pop two Java stack entries.
<code>jit_aastore</code>	Store reference to object array in local variable on Java stack.

Table 1: Examples of JIT library routines.

We now briefly describe how the compiled code accesses data from the VM^{*} runtime. Consider the Jimple expression on line G29 in Figure 5(c) which accesses the `table` array of the `TimeSyncTable` object. The translated C code on line G30 in Figure 5(d) facilitates this through the `jit_getfield` call. Because our class format does not contain the standard constant pool, the application dictionary generated by the registry is used to patch the reference to the `table` field with its two byte encoding (0x8000). Similar operations are required for accessing arguments and returning results at the start and finish of native method execution (e.g., `jit_pop_*` routines).

The generated C file is compiled using an optimizing compiler. C is a natural target language as it facilitates compilation to a wide range of architectures, and the GNU C compiler implements a wide range of optimizations. JIT compiling a Java method means that the class file will need to be modified to mark that method as native by toggling its `ACC_NATIVE` flag. The modified code is thus fed to Grimp, which facilitates the generation of the modified class file. During each stage, Soot applies various optimizations which usually results in a class file that is optimized over the orig-

inal input.

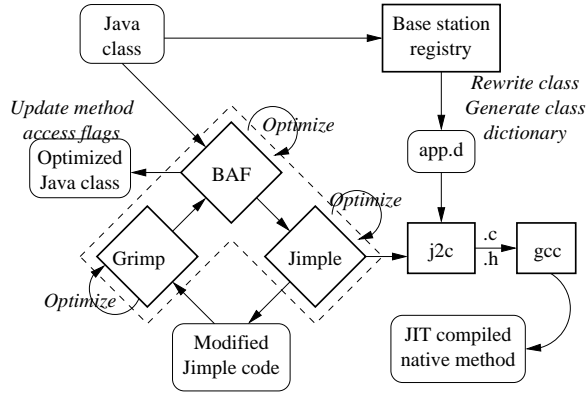


Figure 4: *j2c* transformation stages.

Figure 5 shows the intermediate code generated when JIT compiling a Java method. The example is from the time synchronization service and the excerpt from the `addNewEntry` method is responsible for updating the state table as new messages are received. Soot provides various pre-compiler optimizations to the three-address Jimple code. We disable these in favor of the GNU C compiler optimizations. A useful side-effect of using the Soot framework is that the modified class file that is produced is an optimized version of the original class file.

Note that there are no symbols introduced into the `.data` or `.bss` sections which simplifies linking against the existing VM. The C code closely resembles the three-address Jimple code and is optimized easily by the compiler. For example, Table 3 shows the footprints of the `addNewEntry` method with original bytecode, and when compiled for the ATMega128 and MSP430 CPUs with and without optimization. The figures in the table show a significant difference in footprint between optimized and non-optimized JIT compilation.

The JIT system’s library routines to manipulate the Java stack occur fairly frequently, especially in virtual methods, and inlining them will offer some performance advantage. However, code footprint will increase significantly in doing so. For this reason, we have not marked them for inlining. Some functions such as `jit_pop_one_entry` are very simple and can be inlined. In this case, the Java stack pointer will be directly manipulated by the C code. In fact, a linker can be used to edit the C code to patch in the address of the stack pointer directly, which will result in very efficient object code.

The code samples in Figure 5 show the different patterns of VM-native interaction used by interpretation and JIT compiled code. They also show several benefits of using native code. We discuss these below.

VM ↔ Native argument exchange overhead

Line A in Figure 5(a) involves a call to a virtual method. Although the `is_sync` method does not take explicit arguments, the object on which a virtual (non-static) method is invoked (the `this` object) is an implicit argument. The

`aload_0` bytecode in Figure 5(b) pushes `this` on the Java stack before executing the `invokevirtual` bytecode. In the equivalent C code (lines A12-A13 in Figure 5(d)), JIT library routines that wrap VM’s native interface are used to transfer `this` from the native C stack to the Java stack before switching to the invoked method. (The `jit_invokevirtual` call will result in switching from native mode to interpreted mode if the invoked method is not JIT compiled as well). The `is_sync` method returns a boolean value. In the interpreted version the result is left on the stack by `is_sync`’s bytecode and is consumed by the `ifeq` operation on line A4 in Figure 5(b). The native version needs to transfer the result from the Java stack to a temporary before the native mode can use the result (line A14 in Figure 5(d)). JIT compilation helps reduce Java stack overheads, but data exchanges are unavoidable in hybrid execution and the performance overhead of these exchanges is VM-specific.

Branches

In the JVM, branch bytecodes consume one or more values from the top of Java stack and jump to a different address if a bytecode-specific condition is true. Branches are rather expensive as the interpreter pointer needs to be manipulated. Also, the virtual execution environment is completely detached from the native hardware. Thus, branch prediction mechanisms on some CPUs cannot be used. Dynamic superinstructions and inlined threading techniques improve branch prediction. However, even with these techniques, the target address needs to be fetched from the bytestream. In the C code generated by *j2c*, the target is translated to a C label and a direct jump is made (line A4 in Figure 5(b) and A15 in Figure 5(d)). Further, there is no need for a VM IP in the native mode removing the need for IP manipulations.

Constants

The IP is also needed in interpreters to access constants. There are three kinds of constants (see Figure 5(b)): embedded constants (e.g., line E14), immediate constants (e.g., line B6) that follow the bytecode, and wide constants (e.g., line H36) that are fetched from the class’ constant pool. Embedded constants are implicit in the bytecode and only require pushing the value on the stack. Accessing immediate constants involves fetching bytes from the bytestream to extract the constant value, pushing the value onto the Java stack and incrementing the IP. Accessing wide constants involves fetching a constant pool index from the bytestream, extracting the constant from the constant pool, pushing the constant onto the stack, and incrementing the IP. JIT compiled versions avoid most of these overheads by simply embedding the constant in C code (lines E23, B16, and H34 in Figure 5(d)).

Reducing Java stack overheads

The stack-oriented nature of the JVM results in frequent stack accesses, many of which can be optimized away when translated to register-oriented code. For example, lines G30-G34 in Figure 5(b) push two long integers on top of the Java stack, pop them, add them together, and push the result back on the stack. This is especially costly because JVM stack entries are four bytes wide which results in two pushes or pops when dealing with eight byte long integers. The

equivalent C code in Figure 5(d) subsumes all these stack operations by using register-oriented arithmetic.

The stack computing model also results in repeated pushes of variables that are accessed. For example, the references to array `table` on lines G13 and I15 in Figure 5(a) results in two separate pushes of its containing object (through the `aload_0` bytecode) on lines G28 and I39 in Figure 5(b). Pushing four byte values onto a software stack is more expensive than the corresponding C code which simply reloads register `r0` (lines G30 and I36 in Figure 5(d)) in which the object reference is stored.

An issue specific to the JVM is that stack entries are fixed at four bytes. Thus, even byte variables such as `tableEntries` in Figure 5(a) need to be cast to and from integers when they are accessed by other bytecodes such as `iadd` on line F23 in Figure 5(b). In C, `tableEntries` is assigned to a single register or memory location, resulting in efficient operations in the subsequent references to it.

5.4 Relinking and code distribution

After the modified classes and C files are generated, the C files are compiled without linking. The three address C code generated is inefficient by itself, but can be easily optimized by most compilers. We use the standard GNU C compiler with optimization for size (`-Os`). The resulting object files are linked against the current VM image to produce the new VM. The only references that need to be resolved are calls to the JIT routines that allow the native environment to interact with VM land.

Ideally, an incremental linker [11] should be used to link the JIT compiled routines with the original VM. This will enable very efficient updates to be generated. However, most ordinary linkers are sufficient to perform the necessary linking. For example, if `native.o` is an unlinked binary containing JIT compiled methods and `app.vm.current.exe` is the current application-VM image, the following invocation for the AVR port of the standard GNU linker will generate a fully linked VM delta that can then be distributed:

```
avr-ld native.o -R app-vm-current.exe -o delta
```

The native sources generated by the `j2c` tool do not introduce new symbols to the `.data` or `.bss` sections. The only new symbols added are the native methods themselves, in the `.text` section. Also, there are no direct calls to these methods from the VM. The calls are made indirectly from the application classes, through the native interface. The addresses of the native methods are read from the modified application class that accompanies the JIT compiled binary. Therefore, the delta object is essentially position-independent and can be distributed as is. The size of the distributed object can be reduced by using diff techniques.

The nodes contain a fixed bootloader that receives and commit delta modules to program memory. Code distribution protocols and bootloader designs should be resilient to node failures and power glitches that can potentially reboot a node during a reprogramming operation. After installation, the nodes are reset. Although continuous update models are possible, these involve on-stack replacement and other oper-

ations that lead to complications and compromise program correctness.

6. EVALUATION

With communication being several times more expensive than computation, application writers attempt to increase computation or in-network processing to reduce the communication requirements of the application. JIT compilation is intended to optimize the computation phases. Thus, for evaluation purposes we pay closer attention to the execution speedup of the computation phase. Significant speedups in execution directly translate to decreased CPU active times in a real deployment and thus, prolonged deployment lifetime. We have implemented a number of synthetic applications that involve computation that is fairly intensive for the limited CPUs in this domain. We have also implemented a time synchronization service and evaluated the effect of JIT compilation on some of its core routines. The evaluation has been performed for the Mica2 and Telos platforms.

6.1 Synthetic benchmarks

Successive overrelaxation (SOR) is a method for solving a system of equations, and is derived from the *Gauss-Seidel* (GS) method. The SOR benchmark performs 100 iterations of successive overrelaxation, and uses multi-dimensional arrays and several arithmetic operations. **Crypt** performs the IDEA (International Data Encryption Algorithm) encryption and decryption on a 40 byte array. Crypt also uses arrays and arithmetic operations, but is larger than SOR. **SORT** is a collection of sorting algorithm implementations. These include standard quick sort (QS), an enhanced quick sort (EQS), fast quick sort (FQS), heap sort (HS), extra storage merge sort (ESMS), and shell sort (SS). Each of these are array intensive, and use several comparison operations. Some of them (e.g., quick sort variants) are call intensive. All the sort tests work with an array of 40 elements. **BMHR** is an implementation of the *Boyer-Moore-Horspool-Raita* string search algorithm [25]. It uses several array operations and looping constructs. BMHR takes a pattern as an input and locates matches in a predefined string. It performs some preprocessing on the input pattern, which helps determine how far to skip ahead in the text if an initial match attempt fails. BMHR runs in $O(MN)$ worst case, where M is the pattern length and N is the text length. It has superior average case complexity than several other string search algorithms. In our tests, the pattern input is 8 bytes long and the text is a 129 byte string.

6.2 Time synchronization

We also implemented a time synchronization service in VM^{*}. The Flooding Time Synchronization Protocol (FTSP) [17] allows sensor nodes to synchronize clocks with microsecond accuracy. The network dynamically elects a sensor node as the root node, which other nodes use as a time base. The root node (and other synchronized nodes) periodically broadcast timestamp messages. Upon receiving a message, the node timestamps the message again and adds the value to a fixed-size table of time offsets with an oldest-first replacement policy. The algorithm then attempts to determine the average offset between transmission and reception time by iterating through the valid table entries. This average offset is then used to determine the clock skew, which is

a floating-point multiplier to the time offset. Applications can use this calculated offset and skew to determine a global-time which has up to microsecond accuracy throughout the network. The time synchronization experiments determined execution speedups of processing eight timestamp packets.

6.3 Methodology

Our evaluation focuses mainly on the code speed-memory footprint tradeoff. We used the Avrora framework [27] which provides cycle-accurate simulations of the Mica2 platform. We wrote a custom monitor to extract timing information between pairs of code addresses in benchmark applications. We also used a monitor to report the stack depth of the Java and OS stack (i.e., native C stack). We selected various methods in each application for JIT compilation and measured the corresponding speedups. Avrora does not support the MSP430 CPU used on the Telos platform¹. We also observed the footprints of the compiled modules with different optimization levels. Comparisons are made with the original class footprints. Table 2 enumerates the combinations of methods that we JIT compiled in each of these applications. Where multiple methods are involved, the relevant portions of the call graphs are also shown. For example, `FQS` invokes the `QuickSort` method followed by the `InsertSort`. `QuickSort` calls `swap`, and makes recursive calls to itself. Depending on the input, invocation counts may vary. For consistency across runs, our experiments use a fixed test input set for each application.

In general, determining n (the number of times the candidate portion of bytecode is executed) and j (the fraction of bytecode that is JIT compiled) is not straightforward. Invocation counts do not always give accurate estimates because there may be several portions of the method that are rarely executed, and some portions that are *hot*. For example in our input set, `EQS.brute` is invoked 19 times, but a conditional at the start of the method returns in most invocations. The core of the method is executed only 3 times. In such cases, j is not exactly equal to the ratio of `EQS.brute`'s footprint to the total footprint, and n is not exactly equal to the invocation count.

6.4 Memory footprints

6.4.1 Program memory

Table 3 shows the program memory footprints of the various methods, in bytecode, and native object code compiled with `-Os` and `-O3`. MSP430 clearly has denser binary code, with an average k factor of 3.74 with `-Os` optimization, compared to 5.58 for the ATmega128. Thus, the benefits of using JIT compilation can vary considerably across architectures. The k factors for moderate to large methods are quite consistent across applications for each platform. Deviations from the mean are caused mainly by extremely small methods. For small functions such as `FQS.swap`, k is relatively high. This is because there is very little straight-line code that can be optimized away. For such cases, execution speedups will not be significant. We have not yet explored various optimizations possible with the intermediate representations used in the

¹As we currently do not have access to a logic analyzer, we were unable to report stack depths and timing data for the Telos. These amendments are forthcoming.

	JIT compiled method(s)	Call graph
1	<code>SOR.execute</code>	
2	<code>IDEA.cipher_idea</code>	
3	<code>EQS.sort</code>	
4	<code>EQS.brute</code>	
5	<code>EQS.sort,brute</code>	
6	<code>ESMS.sort</code>	
7	<code>FQS.QuickSort</code>	
8	<code>FQS.swap</code>	
9	<code>FQS.InsertionSort</code>	
10	<code>FQS.QuickSort,swap,InsertionSort</code>	
11	<code>HS.sort</code>	
12	<code>HS.downheap</code>	
13	<code>HS.sort,downheap</code>	
14	<code>QS.sort</code>	
15	<code>SS.sort</code>	
16	<code>BMHR.searchBytes</code>	
17	<code>TimeSync.processMsg, addNewEntry,calculateConversion</code>	

Table 2: Scenarios considered for JIT compilation.

translation tool. It is sometimes possible to avoid the negative effects of small methods using inter-procedural analysis techniques. A design choice that we made in order to reduce footprint was to use calls to the JIT library routines instead of inlining them. Even straight-line code may access objects frequently, generating several calls to `jit.getField`. Table 3 also shows the worst case size of the JIT library routines. We have not yet completed integration of the system with an incremental linking mechanism which would allow us to selectively include only those routines that are needed.

6.4.2 Data memory

Since JIT compiled methods do not introduce any global data, there is no increase in the `.bss` or `.data` footprints. Also, because the natively compiled method allocates the same number of objects or arrays as the original code, there is no increase in dynamic memory requirements. Stack requirements do change however. We extracted the maximum depth of both the native and Java stacks for each experiment, and these are reported in Figure 6. Native stack requirements generally increase, because of the calls to the JIT library routines. The Java stack depth decreases in most cases, because the native method eliminates or reduces most of the stack operations present in the original bytecode. Note that although we report Java stack requirements in bytes, each entry of the stack is four bytes wide. Thus, the 104 byte depth in `SOR` accounts for 26 Java stack entries. The overall increase in memory requirements is mainly due to the fact that there are calls to the JIT library routines. The depth can be decreased by inlining smaller JIT library routines in generated C code. Also, some of the algorithms (e.g., the sorting algorithms) are highly recursive. Native compilation pushes these recursive calls to the native stack which explains the increased OS stack requirements.

6.5 Execution efficiency

To measure execution efficiency, we measured speedups of JIT compiled code with their interpreted counterparts. The timing measurements account for the native call as well.

Java method	Bytecode	Mica2 (ATMega128)				Telos (MSP430)			
		-Os	-O3	k_{-Os}	k_{-O3}	-Os	-O3	k_{-Os}	k_{-O3}
SOR.execute	138	628	664	4.55	4.81	440	440	3.19	3.19
IDEA.cipher_idea	515	2898	2930	5.62	5.69	1656	1694	3.22	3.29
EQS.sort	170	784	772	4.62	4.54	604	610	3.55	3.59
EQS.brute	287	1204	1222	4.19	4.26	966	996	3.37	3.47
ESMS.sort	135	658	648	4.87	4.80	526	520	3.90	3.85
FQS.QuickSort	182	882	840	4.85	4.62	720	734	3.96	4.03
FQS.swap	17	190	190	11.17	11.17	116	116	6.82	6.82
FQS.InsertionSort	67	318	324	4.75	4.84	220	232	3.28	3.46
HS.sort	59	346	422	5.86	7.2	274	318	4.64	5.39
HS.downheap	80	406	634	5.08	7.93	286	302	3.58	3.78
QS.sort	200	886	842	4.43	4.21	662	670	3.31	3.35
SS.sort	102	438	452	4.29	4.43	276	280	2.71	2.74
BMHR.searchBytes	142	586	580	4.13	4.08	390	390	2.76	2.76
TimeSync.processMsg	27	224	224	8.29	8.29	148	148	5.48	5.48
TimeSync.addNewEntry	237	1404	1112	5.92	4.69	874	862	3.69	3.64
TimeSync.calculateConversion	476	3164	3114	6.65	6.54	1178	1164	2.47	2.45
JIT native library	-	8034	8588	-	-	6012	6642	-	-

Table 3: Code footprints and k (bloat factor) on Mica2 (ATMega128) and Telos (MSP430).

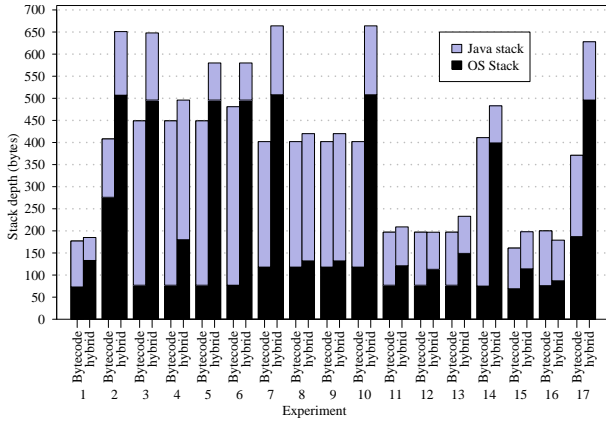


Figure 6: Effects of JIT compilation on Java and native stacks (ATMega128 with -Os).

Comparisons were made with the non-threaded interpretation mode in VM[★].

Figure 7 shows the speedups observed for the experiments in Table 2 with -Os optimization. The speedups are quite significant in most cases. Larger method bodies containing frequent computations will yield even greater speedup. The experiments that yielded lower speedup are either due to small methods getting JIT compiled, or due to insufficient m values (i.e., execution coverage). In 4, EQS.brute is invoked 19 times, but its core loop is executed only 3 times. In 8, FQS.swap is an extremely small method that is called frequently (56 times). The increased speed of the native version is offset by the high switching cost between interpreted and native mode. In 9, for the particular input that we use, FQS.InsertionSort is only invoked once leading to only a modest increase in performance.

There are *switching overheads* involved when moving between native and interpreted mode through calls or returns. Arguments need to be transferred from the Java stack to C variables, and return values need to be copied from C

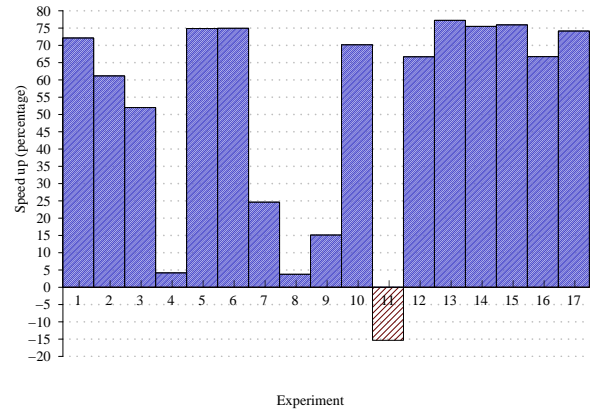


Figure 7: Percentage execution speedup (ATMega128 with -Os).

variables to the Java stack. Calls need to create a new Java stack frame for the method invocation as well. The most expensive calls are from JIT compiled methods to non-native methods. In addition to the above overheads, a new interpreter incarnation is necessary for correct continuation after returning from the method. This explains the negative effect of JIT compiling the `sort` method of HS. This method is fairly small and repeatedly calls the `downheap` method. The native speedup of `sort` is insufficient to offset cost of switching to the interpreted `downheap` method. In fact, JIT compiling `downheap` by itself results in significant speedup. In these circumstances, it makes sense to apply JIT compilation recursively to the methods called from candidate methods. This is clear from the speedup in the case when both `sort` and `downheap` are JIT compiled. Inside a native method itself, there are overheads due to accesses to objects in the VM data space. The costs of some of these operations in cycles on the ATMega128 are as follows: `jit_getfield_four_bytes` (142), `jit_pop_one_entry` (20), `jit_iastore` (197), `jit_arraylength` (248).

JIT compilation is more effective in non-trivial methods

which have high interpretation overhead (e.g., straight line code) with lower auxiliary overheads due to operations such as object allocation, method invocations, etc. In other scenarios such as call intensive methods, optimizations such as object inlining could help.

7. RELATED WORK

The primary focus of this paper is on developing a hybrid execution environment that provides the benefits of interpretive and native modes for WSN applications. In the WSN domain, operating systems (such as TinyOS [10], MOS [1], and SOS [7]) have been the primary system software layer for developing and running WSN applications. These applications run in the native mode. The authors are aware of only two approaches — Maté [13, 14] and VM[★] — that support running WSN programs in the interpretive mode. VM[★] is currently the only VM-based approach that implements an adaptive compilation framework for supporting hybrid execution. In the rest of this section, we briefly describe related work that use various JIT compilation schemes, support distributed compilation framework, or use similar profiling techniques.

Compilation schemes include techniques for eliminating the cost of interpretation by converting bytecode into binary. Toba [24] performs this translation before the application is executed. JIT compilers (such as Sun’s HotSpot JVM [26] and IBM’s Jalapeno JVM [2]) identify performance critical methods dynamically and compile them while the method is running. The continuous compilation scheme [21] interleaves the executions of interpreted code, native code and compilation. It is based on the key insight that code for a method should be ready before it is called. The VM[★] compilation framework uses a simpler form of JIT compilation: unlike most JIT approaches that continuously monitor, update and compile hot methods, VM[★] performs these functions only once. Much of this decision is driven by the resource constraints of sensing devices. A key goal in the entire VM[★] JIT compilation process is to keep the parameter j low so that only performance critical methods are compiled and distributed.

Distributed compilation frameworks focus on migrating the tasks of profiling, compiling, and linking to a different host. Approaches differ in the level of this migration and the architecture used to achieve the migration. [30] proposes a framework in which portions of code are compiled at a server. The server uses runtime data and characteristics to compile and send optimized code to the client. The framework assumes a tightly coupled client and server environments. JCOD [5] proposes a remote compilation framework in which a server compiles an OS and a VM-independent native code for embedded devices. The clients adapt the code for a specific platform.

The VM[★] compilation framework is similar to the above approaches in that it migrates the compute and memory-intensive compilation task to a base station. The differences arise in the scope of compilation (small number of critical methods), the nature of code generated (small size), and the interface support provided by the VM[★] runtime.

Distributed profiling involves collecting profiling data from

remote systems. Several systems use the notion of distributed profiling. For instance, many commercial operating systems (e.g., Windows XP) and applications (e.g., Microsoft Office, GNOME and KDE) collect crash data; [19] collects test coverage data; [15, 8] mine remote profiling data for bug isolation; and [18] collects profiling data to evaluate application performance. Much of the effort here is on the scope of sampling and analysis of the sampled data.

The VM[★] JIT compilation framework implements a lightweight distributed profiling system for collecting data about performance critical methods. The distributed profiling problem in the WSN domain differs significantly due to several reasons: First, the data collection needs to be very lightweight — both in terms of size and the overhead of instrumentation. Nodes do not have memory resources to store large amounts of profiling data. Second, since the cost of sending messages is high, the sampling data needs to be aggregated fairly aggressively. Finally, since nodes execute identical applications, the profiling data is fairly similar. It means that nodes can use a sampling scheduling scheme to decide which methods to compile. The approach described here focuses more on efficient ways of collecting and disseminating data, and less on analyzing the collected data.

8. CONCLUSIONS AND FUTURE WORK

With increasing attention being given to VMs as system software for sensor networks, optimizing their execution is an important goal. Our work shows that focusing on more efficient interpreters, bytecode formats and OS schedulers has fundamental limits. Performance can be improved in most cases through the hybrid execution of VM and native code.

There are several research issues that we plan to explore in the future. As mentioned in Section 6, invocation counts are not ideal estimators for j and n . Method execution time and method-level coverage information is needed to make better estimates to select methods for JIT compilation. Trace information in combination with static analysis could be used to make more accurate estimates of these quantities. Another observation we obtain from the analytical framework is that the benefits are quite sensitive to the choice of code that is compiled. Some methods contain rarely executed code that ideally should not be JIT compiled. Thus, increasing the granularity of JIT compilation is likely to yield much better performance-footprint tradeoffs than the current approach of JIT compiling entire methods, by offering finer control over j . In [16], a patented approach for fine-grained JIT compilation using a tiny JIT compiler with a footprint of about 10KB is described. Although the compiler they use is limited in the code optimizations it implements, it opens up the possibility of in-network JIT compilation which furthers the case for hybrid execution models since the code distribution cost can be decreased even further. A higher end sensor node could be assigned to JIT compile code for a cluster. We are also examining various low overhead instrumentation schemes, and the usefulness of allowing application level access to select routines for providing hints to the selection mechanism.

9. REFERENCES

- [1] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. MANTIS: System support for multimodal networks of in-situ sensors. In *Proceedings of the 2nd ACM International Workshop on Wireless Sensor Networks and Applications (WSNA) 2003*, pages 50–59, 2003.
- [2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices*, 35(10):47–65, 2000.
- [3] J. Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003.
- [4] J. R. Bell. Threaded Code. *Communications of the ACM*, 16(6):370–372, 1973.
- [5] B. Delsart, V. Joloboff, and E. Paire. JCOD: A Lightweight Modular Compilation Technology for Embedded Java. In *EMSOFT '02: Proceedings of the Second International Conference on Embedded Software*, pages 197–212, London, UK, 2002. Springer-Verlag.
- [6] M. Ertl. Stack Caching for Interpreters. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 315–327, La Jolla, CA, 1995. ACM Press.
- [7] S. Han, R. Rengaswamy, R. S. Shea, E. Kohler, and M. B. Srivastava. A dynamic operating system for sensor nodes. In *Third International Conference on Mobile Systems, Applications and Services*, page 14, Seattle, WA, June 2005. USENIX/ACM.
- [8] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 156–164, New York, NY, USA, 2004. ACM Press.
- [9] J. Hill and D. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, November/December 2002.
- [10] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.
- [11] J. Koshy and R. Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *Proceedings of the second European Workshop on Sensor Networks (EWSN)*, 2005.
- [12] J. Koshy and R. Pandey. VM*: Synthesizing scalable runtime environments for sensor networks. In *Proceedings of the third international Conference on Embedded Networked Sensor Systems (Sensys)*, San Diego, CA, USA, Nov 2005. ACM.
- [13] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *Proceedings of the Eleventh Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–95. ACM Press, Oct. 2002.
- [14] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *Proceedings of the 2nd USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2005.
- [15] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 141–154, New York, NY, USA, 2003. ACM Press.
- [16] G. Manjunath and V. Krishnan. A Small Hybrid JIT for Embedded Systems. *ACM SIGPLAN Notices*, 35(4):44–50, 2000.
- [17] M. Maróti, B. Kusy, G. Simon, and A. Lédeczi. The Flooding Time Synchronization Protocol. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 39–49, Baltimore, MD, USA, 2004. ACM Press.
- [18] P. Nagpurkar, C. Krintz, and T. Sherwood. Phase-aware remote profiling. In *International Symposium on Code Generation and Optimization (CGO05)*, San Jose, CA, 2005.
- [19] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma system: continuous evolution of software after deployment. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 65–69, New York, NY, USA, 2002. ACM Press.
- [20] I. Piumarta and F. Ricciardi. Optimizing Direct Threaded Code by Selective Inlining. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, Montreal, Quebec, Canada, 1998. ACM Press.
- [21] M. P. Plezbert and R. K. Cytron. Does just in time = better late than never? In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 120–131, New York, NY, USA, 1997. ACM Press.
- [22] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks: Special track on Platform Tools and Design Methods for Network Embedded Sensors (IPSN/SPOTS)*, 2005.
- [23] T. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 322–332. ACM Press, 1995.
- [24] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. W. C. J. 1997. Toba: Java for applications, a way ahead of time (wat) compiler. In *Third USENIX Conference on Object-Oriented Technologies*. USENIX, June 1997.
- [25] T. Raita. Tuning the Boyer-Moore-Horspool String Searching Algorithm. *Software - Practice and Experience (SPE)*, 22(10):879–884, 1992.
- [26] SUN Microsystems. The Java HotSpot Virtual Machine 2.0, June 2000.
- [27] B. Titzer and J. Palsberg. Nonintrusive Precision Instrumentation of Microcontroller Software. In *Proceedings of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 59–68, Chicago, IL, June 2005.
- [28] L. Ton, L. Chang, M. Kao, H. Tseng, S. Shang, R. Ma, D. Wang, and C. Chung. Instruction Folding in Java Processor. In *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS)*, pages 138–143, Seoul, Korea, 1997. IEEE Computer Society.
- [29] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - A Java Optimization Framework. In *Proceedings of Centers for Advanced Studies Conference*, pages 125–135, 1999.
- [30] M. J. Voss and R. Eigenmann. A framework for remote dynamic program optimization. In *DYNAMO '00: Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, pages 32–40, New York, NY, USA, 2000. ACM Press.
- [31] J. Wu and R. Pandey. BOTS: A constraint-based component system for synthesizing scalable software systems. Technical Report TR-CSE-2005-18, University of California, Davis, Aug. 2005.

```

1: package edu.ucdavis.senses.app.jit;
2:
3: public class TimeSyncTable {
4:     public void addNewEntry(TimeSyncMsg msg) {
5:         ...
6:         if (is_synced() &&
7:             (timeError > ENTRY_THROWOUT_LIMIT ||
8:              timeError < -ENTRY_THROWOUT_LIMIT))
9:             clearTable();
10:
11:         for (i = 0; i < MAX_ENTRIES; i++) {
12:             ++tableEntries;
13:             age = msg.arrivalTime - table[i].localTime;
14:             if (age >= 0x7FFFFFFF)
15:                 table[i].state = 0;
16:             ...
17:         }
18:     }
19: }
20: ...
21: }

```

(a) Java source code

```

1: ...
A 2: aload_0
A 3: invokevirtual #18; //Method is_synced():Z
A 4: ifeq 71
B 5: iload 9
B 6: sipush 10000
B 7: if_icmpgt 67
C 8: iload 9
C 9: sipush -10000
C 10: if_icmpge 71
D 11: aload_0
D 12: invokevirtual #4; //Method clearTable():V
13:
E 14: iconst_0
E 15: istore_2
E 16: iload_2
E 17: bipush 8
E 18: if_icmpge 173
F 19: aload_0
F 20: dup
F 21: getfield #9; //tableEntries:B
F 22: iconst_1
F 23: iadd
F 24: i2b
F 25: putfield #9; //tableEntries:B
G 26: aload_1
G 27: getfield #15; //edu.ucdavis.senses.app.jit.TimeSyncMsg.arrivalTime:J
G 28: aload_0
G 29: getfield #3; //table:[Ledu.ucdavis.senses.app.jit.TimeSyncTableEntry;
G 30: iload_2
G 31: aaload
G 32: getfield #7; //edu.ucdavis.senses.app.jit.TimeSyncTableEntry.localTime:J
G 33: lsub
G 34: lstore 5
H 35: lload 5
H 36: ldc2_w #19; //long 2147483647L
H 37: lcmp
H 38: iflt 125
I 39: aload_0
I 40: getfield #3; //table:[Ledu.ucdavis.senses.app.jit.TimeSyncTableEntry;
I 41: iload_2
I 42: aaload
I 43: iconst_0
I 44: putfield #6; //edu.ucdavis.senses.app.jit.TimeSyncTableEntry.state:B
45: ...

```

(b) Java bytecode

```

1: edu.ucdavis.senses.app.jit.TimeSyncTable r0;
2: edu.ucdavis.senses.app.jit.TimeSyncMsg r1;
3: byte b0, b1, b5, b14, b16, b20, b21, b22, b24, b25, b27, b29;
4: long l2, l3, l16, l17, l18, l10, l11, l12, l17, l18, l19, l30, l31, l32, l33;
5: int i4, i9, i13, i15, i23, i26, i28, i34;
6: boolean z0;
7: edu.ucdavis.senses.app.jit.TimeSyncTableEntry[] r3, r5, r7, r11, r13, r15;
8: edu.ucdavis.senses.app.jit.TimeSyncTableEntry r4, r6, r8, r12, r14, r16;
9:
10: r0 := @this: edu.ucdavis.senses.app.jit.TimeSyncTable;
11: r1 := @parameter0: edu.ucdavis.senses.app.jit.TimeSyncMsg;
12: ...
A 13: z0 = virtualinvoke r0.<TimeSyncTable: boolean is_synced()>();
A 14: if z0 == 0 goto label1;
B 15: if i13 > 10000 goto label0;
C 16: if i13 >= -10000 goto label1;
D 17: label0:
D 18: virtualinvoke r0.<TimeSyncTable: void clearTable()>();
19:
E 20: label1:
E 21: b5 = 0;
E 22: label2:
E 23: if b5 >= 8 goto label6;
F 24: b14 = r0.<TimeSyncTable: byte tableEntries>;
F 25: i15 = b14 + 1;
F 26: b16 = (byte) i15;
F 27: r0.<TimeSyncTable: byte tableEntries> = b16;
G 28: l17 = r1.<TimeSyncMsg: long arrivalTime>;
G 29: r3 = r0.<TimeSyncTable: TimeSyncTableEntry[] table>;
G 30: r4 = r3[b5];
G 31: l18 = r4.<TimeSyncTableEntry: long localTime>;
G 32: l19 = l17 - l18;
H 33: b20 = l19 cmp 2147483647L;
H 34: if b20 < 0 goto label3;
I 35: r5 = r0.<TimeSyncTable: TimeSyncTableEntry[] table>;
I 36: r6 = r5[b5];
I 37: r6.<TimeSyncTableEntry: byte state> = 0;
38:
49: label3:
40: r7 = r0.<TimeSyncTable: TimeSyncTableEntry[] table>;
41: ...

```

(c) Generated Jimple code

```

1: void Java.edu.ucdavis.senses.app.jit.TimeSyncTable.addNewEntry(void) {
2:     s1 b0,b1,b5,b14,b16,b20,b21,b22,b24,b25,b27,b29;
3:     s4 i4,i34,i9,i28,i13,i23,i26,i15;
4:     s8 l2,l3,l16,l7,l8,l10,l11,l31,l32,l33,l18,l30,l19,l12,l17;
5:     u1 z0;
6:     ARRAY_REF r3,r5,r7,r11,r13,r15;
7:     OBJECT_PTR r16,r14,r12,r0,r1,r4,r6,r8;
8:
9:     r1 = (OBJECT_PTR)jit_pop_one_entry();
10:    r0 = (OBJECT_PTR)jit_pop_one_entry();
11:    ...
A 12:    jit_push_one_entry((u4) r0 );
A 13:    jit_invokevirtual( r0,4 );
A 14:    z0 = (u1)jit_pop_one_entry();
A 15:    if( z0 == 0 ) goto label1;
B 16:    if( i13 > 10000 ) goto label0;
C 17:    if( i13 >= -10000 ) goto label1;
18:    10:
D 19:    jit_push_one_entry((u4) r0 );
D 20:    jit_invokevirtual( r0,0 );
21:
22:    label1:
E 23:    b5 = 0;
24:    label2:
E 25:    if( b5 >= 8 ) goto label6;
F 26:    b14 = (s1)jit_getfield_one_byte( r0,0 );
F 27:    i15 = b14 + 1; b16 = (s1)i15;
F 28:    jit_putfield_one_byte( r0,0,b16 );
G 29:    l17 = (s8)jit_getfield_eight_bytes( r1,49169 );
G 30:    r3 = (ARRAY_REF)jit_getfield_four_bytes( r0,32786 );
G 31:    r4 = jit_aaload( r3,b5 );
G 32:    l18 = (s8)jit_getfield_eight_bytes( r4,49153 );
G 33:    l19 = l17 - l18;
H 34:    b20 = ((l19<2147483647L)?1:(l19>19)?-1:0);
H 35:    if( b20 < 0 ) goto label3;
I 36:    r5 = (ARRAY_REF)jit_getfield_four_bytes( r0,32786 );
I 37:    r6 = jit_aaload( r5,b5 );
I 38:    jit_putfield_one_byte( r6,0,0 );
39:    label3:
40:    ...

```

(d) Generated C code

Figure 5: JIT compilation stages (line number prefixes indicate matching code segments).