VM★: Synthesizing Scalable Runtime Environments for Sensor Networks[*]

Joel Koshy
Department of Computer Science
University of California, Davis
Davis, California 95616
koshy@cs.ucdavis.edu

Raju Pandey
Department of Computer Science
University of California, Davis
Davis, California 95616
pandey@cs.ucdavis.edu

ABSTRACT

Sensor networks are being deployed at massive scales, containing a range of platforms. Programming paradigms for sensor networks should meet the attendant challenges of scale and heterogeneity. Researchers have considered virtual machines as a means to address these challenges. However, in order to satisfy the resource limitations of sensor nodes, they export only a minimal set of services to the application programmer. This makes applications of even moderate complexity difficult to implement. We present VM★ — a framework for building resource-efficient virtual machines that scale and export comprehensive service suites on a per-application basis. We advocate the use of fine-grained software synthesis to build resource-efficient system software, and facilitate both application changes and system software upgrades at runtime through an efficient incremental update scheme. We have used our framework to build virtual machines on the Mica platform and describe how virtual machines are effective in meeting the difficult demands of heterogeneity and reprogrammability.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems — Network operating systems; C.2.1 [**Computer Communication Networks**]: Network Architecture and Design — Wireless communication

General Terms

Design, Experimentation, Languages, Measurement, Performance

Keywords

Wireless Sensor Networks, Virtual Machines, Operating Systems, Software Synthesis, Programming Languages, Network Reprogramming

[*]This work is supported in part by NSF grants CNS-0435531 and EIA-0224469.

1. INTRODUCTION

Wireless sensor networks (WSN) show considerable promise in bringing the vision of ubiquitous computing [26] to reality. There is growing interest in their use for a variety of applications, and researchers are constantly aiming toward developing smaller and better hardware platforms. It is quite likely that as the field matures, we will see a wide variety of highly efficient hardware specialized to perform specific tasks. Even today, there are several platforms in popular use for research [9]. While this yields significant benefits from an overall system design viewpoint, it throws considerable burden on the programmer who needs to be aware of the idiosyncrasies of each platform. Most of the variation across these platforms is in the implementation of subsystems; e.g., different processors, radio, and sensing modules. The actual operational patterns are more or less uniform, and having to rewrite applications for multiple platforms is redundant and burdensome.

The correct choice of system software used in application development is critical to manage the complexity in this regime. We use the term system software to describe software such as operating systems (OS), virtual machines (VM), and middleware, which export services for applications to build upon. The main challenges to deal with are: *(i) heterogeneity of end systems, (ii) allowing dynamic updates in deployed software,* and *(iii) exporting a rich programming interface while respecting the resource constraints of end devices*. There are inherent tradeoffs involved in meeting these challenges. For example, platform-independence is usually achieved through layers of abstraction, which inevitably compromises performance and resource constraints. Existing OS solutions are mostly special-purpose system software optimized for specific sensor platforms. These approaches are designed for performance, but are unsatisfactory with regard to interoperability and reprogrammability. Thus, existing OS and VM system software address only a subset of these challenges.

In this paper, we present VM★ — a software framework for synthesizing runtime environments for WSNs, keeping the key challenges in focus. The approach is VM-based: programmers write applications over a common abstract interface. Device-specific features are accessed through a lightweight native interface. Porting the VM across various node architectures allows applications to be deployed uniformly in heterogeneous environments.

There are several advantages to using a VM-based approach in WSN applications. First, hardware abstraction allows applications to run transparently over the varied architectures of WSN nodes, so programs do not need to be rewritten for different architectures. Second, VMs can use well-designed instruction sets for specific application domains. This allows rapid prototyping of highly compact application binaries which can be distributed in the network with low energy overheads. The VM instruction set becomes the basis for code sharing, interoperability, and application binary distribution and management. Third, using platform-independent code facilitates the application of basic distributed computing paradigms such as mobile agents and in-network processing.

While the concept of an abstract machine is appealing, WSN application and domain characteristics impose significant challenges on the nature and scope of the VM that can be realized. The first challenge arises from the extreme resource variability among sensor platforms. At one extreme, Mica, Telos, and EYES platforms [9, 18] have fairly slow CPUs with memory resources in the order of kilobytes. Higher end platforms such as the Stargate have faster CPUs and more memory. Thus, a suitable methodology is needed for implementing VMs on a wide variety of devices. The second challenge involves long term management of applications after they have been deployed. All software artifacts change over their lifetime — to add capabilities, fix bugs, etc. Since sensing devices may be embedded in physical settings that are hard to reach, the ability to dynamically update deployed applications and system software is necessary. The third challenge comes from the requirement that WSN applications must be energy efficient. VM-based applications tend to be slower, as they use an interpreter-based execution engine. For VMs to be viable as WSN system software, they need to be made sufficiently efficient in operation.

VM* is based on the key insight that the VM that actually runs on a specific device does not need to reflect the full VM specification. It only needs to provide services that are needed by the application running on the device. Further, different services can be implemented differently on devices based on resource availability. VM* contains a general description of a VM which is instantiated and specialized for each application and each device. It also implements a continuous update model in which WSN nodes can be updated incrementally when changes occur in applications and the VM. By tracking program changes at the VM abstraction level, the cost of distributing and applying application updates is significantly reduced. Costs of corresponding updates in the system software are also low due to incremental updates.

The VM* framework runs on the Mica family of nodes, with ongoing work on the Telos, XYZ, Stargate and handheld devices. Our current implementation includes a component language for representing system software components [13], tools for analyzing and compacting Java classes, a component-based OS [19], a component-based implementation of a subset of the Java Virtual Machine (JVM) [17], tools for synthesizing the VM and underlying OS, and an incremental linker [14] to facilitate energy-efficient updates. Application programmers can write programs in Java, and access I/O and sensing devices through native interfaces. The framework is not tightly coupled to either the Java language or the JVM, and the concepts can be applied to other languages and VMs. Indeed, one of our broader research goals is to explore the VM design space in WSN applications. The framework provides us with the necessary software infrastructure and tools to do so. Our results show that we can construct a range of VMs with different resource requirements and capabilities. The runtime performance behavior of Java applications, while lower than native implementations, is acceptable. We have implemented various optimizations that reduce the runtime overheads of the VM.

The paper is organized as follows. Section 2 surveys related work, and highlights VM*'s differentiating characteristics. In Section 3, we give a conceptual overview of our framework and its key components. Section 4 provides implementation details. Our evaluation is presented in Section 5, and Section 6 contains conclusions and future work.

2. RELATED WORK

Existing system software for WSNs can be classified into two main approaches: (i) OS-based approaches such as TinyOS [10], SOS [8], Mantis [1], and PEEROS [18], and (ii) VM-based approaches such as Maté [15]. Due to space constraints, we focus on TinyOS, Maté and SOS.

TinyOS [10] is a component-based OS developed at UC Berkeley. It is characterized by an event-driven programming model well-suited for most WSN applications. Applications are written in nesC [7], and are defined by wiring together user-defined and library components. Wirings specify dependencies and bind implementations to abstract components. Concurrency is supported through split-phase non-blocking execution. Short-running atomic computations can post long-running tasks that run to completion if not preempted by events. The scheduler puts the processor to sleep when there are no pending tasks in the task queue. The component-based programming model facilitates reuse of software and OS components, and the generation of compact application-OS composite binaries. Although reprogrammability is not a major consideration in its design, the Deluge component [11] allows over-the-air whole system reprogramming after deployment by distributing new application-OS composite binaries.

Maté [15] is a stack oriented VM implemented using TinyOS. It is built over several system components providing access to sensors, transceivers, and external storage. Maté instructions hide the asynchronous nature of the TinyOS event model to provide a simpler synchronous programming interface. It implements a fixed thread-pool of *contexts* that react to hardware events and commands from the application. Each context has its own operand stack for passing data between operations. Contexts can share variables, and run concurrently at instruction granularity. Each instruction is executed as a TinyOS task. The instruction set is designed for compactness. For example, frequently used opcodes contain embedded operands. Users can also define custom instructions for a domain-specific instruction set. A key goal of Maté is reprogrammability. VM applications are injected as code capsules into deployments of nodes programmed with Maté. A viral code distribution scheme infects nodes in the network with the application capsules. More recent work on Maté generalizes the framework for building application specific VMs [16].

SOS [8] is an OS with a small kernel, and uses dynamically loadable modules to facilitate software evolution. It features a flexible priority-based scheduling system that queues messages between modules for future execution. Applications are composed of a set of modules interacting via asynchronous messages, or indirect function calls made through function handles obtained from function control blocks. SOS also provides a simple dynamic memory subsystem. In terms of reprogrammability, it offers a middle-ground between the TinyOS-Deluge combination and Maté. With Deluge, entire binaries are distributed at high cost. While Maté's code distribution is energy-efficient due to small code capsules, the system software itself cannot be changed after deployment. SOS allows energy-efficient updates by reprogramming at module granularity. Modules can be loaded and removed on the fly, with any necessary updates made to the function control blocks.

VM* differs from the above approaches in two main ways. First, it contains a rich service layer which allows programmers to write applications easily. Software synthesis is used to precisely tailor and scale the system software to the needs of each application, within the constraints of end devices. The current implementation of Maté is more domain-specific, although similar principles can be applied to make it genuinely application-specific. Second, the update model is flexible and low cost. Maté allows updates of VM applications, but the VM itself cannot be changed easily after deployment. Our incremental linking feature adds flexibility to the system by allowing fine-grained updates to both the VM application and the system software. It is more flexible than the scheme used in SOS which is based on indirection. To avoid indirect function calls, we by use an enhanced linker that patches calls to updated functions directly.

3. THE VM★ FRAMEWORK

VM★ is a software framework for building and maintaining runtime environments for WSN applications. A key design goal has been to develop programming languages and tools that allow us to capture components and relationships among the components of system software at very fine granularity. Finer granularity of system components yields higher control over system construction and updates. The components and relationships between them are utilized to identify, instantiate, and update specific components. At its core, the framework features a component language to represent software systems, a composition tool for selecting and composing specific components, and infrastructure for updating applications and system software.

We begin this discussion by introducing the programming models available in the framework. We then describe how the application, device, and component specifications are used to synthesize a runtime environment. Finally, we describe the incremental update mechanism which enables dynamic adaptation of the statically configured runtime environment.

3.1 Programming model and concurrency

In this section, we briefly describe how WSN applications can be written using the VM★ framework. The programming models we discuss here do not require any extensions to the Java programming language. Programs import natively implemented Java libraries for accessing hardware functions such as sensing and communication. We see these models as enablers for more domain-specific programming models.

There is considerable variation in the use cases and degree of concurrency in WSN applications. Some applications are specialized to perform a dedicated sensing task. For example, a node may periodically sample its sensor and maintain a running average over time, transmitting averages at regular intervals. At the system software level this requires various handler routines for events such as communication and timer events. Thus, system software service threads[1] will be running, but from the application programmer's view an imperative programming style with synchronous calls to native services is sufficient and natural to use. In the synchronous programming model, much of the concurrency is hidden from the application, and is implemented through blocking calls. For example, periodic sampling of a sensor can be implemented through synchronous sense and sleep calls.

More complex applications require custom event handling for reacting to external stimuli and radio events. For example, in a data aggregation service, nodes may need to process or filter incoming data packets instead of simply forwarding them up the aggregation tree. In some applications, nodes may adaptively form clusters due to dynamic nature of link qualities or mobility of target phenomena. In general, such cooperative applications require the ability to have custom event handling and application level threads of control in addition to lower layer services that run concurrently. Even in minimally cooperative applications, lower layer services can benefit from application level control. In all these cases, the programming model should support event detection and handling. VM★ currently provides two event-based programming approaches. The first is a *select* model in which the application can register interests in a number of events. The application then blocks on a select call. When one of the registered events occurs, the call returns and event masks are used to determine which event handler to execute. This is similar to the UNIX select call. The second is an *action listener* model in which the native event handlers invoke callbacks to perform application specific handling. The applications described in this paper use the select interface.

To illustrate their use, consider an application that periodically reads a sensor, and needs custom photo sensor and radio handling. Figure 1 shows how the select call is used to handle these events. The application subscribes to the timer, photo sensor, and radio events, and acquires a select handle. After initialization, the application blocks on the select call until at least one of the events occurs. The appropriate handling method is then invoked for application specific event handling. These methods make calls to the native layer to fetch any native values such as the sensor reading or packets received by the radio. Figure 2 shows the same functionality with action listeners. Application specific event handling classes are extended from default handlers in library classes. For clarity, only the relevant portions are shown. When an event occurs, the native event handler invokes the registered callback method. Although we have used Java 1.5 generics [4], the native layer can be modified to work without generic support.

```
char eList, eVector;
byte sHandle;
eList = Select.setEventId(eList, Events.PHOTOSENSOR | Events.RADIO_RECV |
                                 Events.TIMEOUT);
sHandle = Select.requestSelectHandle();
...
while (true) {
  ...
  eVector = Select.select(sHandle, eList); // blocking call
  if (Select.eventOccurred(eVector, Events.PHOTOSENSOR)) {
    handlePhotoSensor();
  }
  if (Select.eventOccurred(eVector, Events.RADIO_RECV)) {
    handlePacket();
  }
  ...
}
```

Figure 1: Using blocking select calls to handle events.

The select call offers a centralized control point for dispatching handlers, while action listeners offer decentralized control through implicit callbacks. The select interface can block against multiple events but executes event handlers sequentially. Action listeners can allow concurrent event handling provided multithreading support is in place. Each event handling thread blocks until the event occurs and then handles it. Threads need to be scheduled properly to reduce contention and avoid race conditions. Although our underlying OS supports multithreading, the current implementation of the VM only supports single threaded Java applications.

3.2 Specification and composition of system software

Fine-grained specification of the system software is an important first step in the overall process of generating a runtime environment for a wide variety of devices, especially those that have limited resources. A commonly used technique to build customized software systems is through the use of components. The components capture specific services exported and imported by a software entity. A software system can be defined by a set of components and dependencies among components. The second step in the process involves using the specification to generate a system software stack

[1]In this discussion, we refer to threads as control flows that may be interleaved or executed in sequence.

```
package System;

abstract class HWEvent<T extends EventDataCollector> {
  public byte eventId;
  abstract public void hwCallBack(T ed);
}

class HWEventHandler<T extends HWEvent> {
  public HWEventHandler(T obj) {
    addHWListener(obj);
  }
  public native void addHWListener(T obj);
}

final class PhotoCollector extends EventDataCollector {
  private final char photoValue;
  public PhotoCollector() {
    photoValue = 0;
  }
  public char getValue() {
    return photoValue;
  }
}

abstract class PhotoSensorEvent extends HWEvent<PhotoCollector> {
  public PhotoEvent() {
    this.eventId = Events.PHOTOSENSOR;
  }
}
```

```
package Applications;

class AppPhotoSensorEvent extends PhotoSensorEvent
{
  char photoValue;
  public void hwCallBack(PhotoCollector c) {
    this.photoValue = c.getValue();
    { /* custom handling code */ }
  }
}

public class Application {
  public static void main(String argv[]) {
    PhotoSensorEvent photoE = new AppPhotoSensorEvent();
    HWEventHandler<PhotoSensorEvent> photoH =
       new HWEventHandler<PhotoSensorEvent>(photoE);

    RadioRecvEvent rrE = new AppRadioRecvEvent();
    HWEventHandler<RadioRecvEvent> rrH =
      new HWEventHandler<RadioRecvEvent>(rrE);

    TimeoutEvent tE = new AppTimeoutEvent();
    HWEventHandler<TimeoutEvent> tH = new HWEventHandler<TimeoutEvent>(tE);

    while (true) {
      Clock.sleep();
  ...
}
```

Figure 2: Using action listeners to handle events.

for a specific application-device pair. A composition tool reifies the software system by determining dependencies, and instantiating and composing the components. For the approach to be effective, components should be fine-grained, and the composition tool should be capable of selecting various components on the basis of their suitability for specific platforms.

We have designed and implemented a component language called BOTS [13] in which components also include the notion of *attributes*. Attributes specify properties of components, and are instantiated with different values in the actual implementations. For example, a GC component can have an attribute specifying the object header overhead. Attributes drive the selection of specific components during a composition process. A software system is built by defining a set of components, connections among components, and conditions under which specific components can be selected. The composition tool then analyzes the components, determines dependencies among the components, evaluates the constraints on the attributes, and selects those components that satisfy specific constraints.

Figure 3 shows how synthesis and scaling are used to build a runtime environment. The application binary $app.class$ is analyzed to determine the configuration of a runtime environment for $app.java$. The software synthesis and scaling tool uses the application specification, component descriptions, and target node characteristics to build an annotated component dependency graph. The

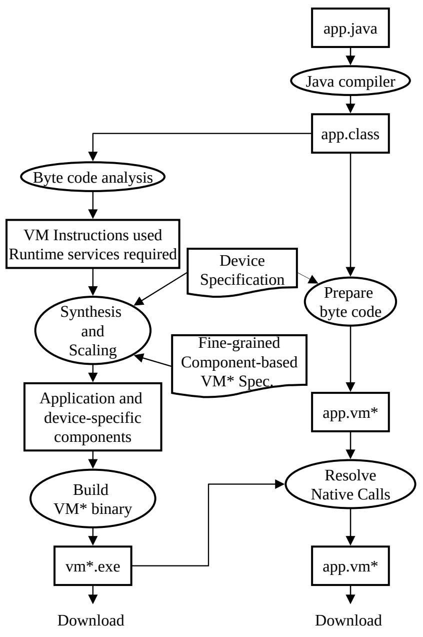


Figure 3: Synthesizing a VM for a device and application.

tool then selects all components that satisfy specific constraints and builds a VM binary. During the process, a binary image map stores various information such as addresses of symbols, sizes, etc., and is used to patch native calls made by the VM application. The nodes are then programmed with the generated system software and VM application.

3.3 Application and system software evolution

After deployment, application requirements may change or software enhancements may need to be made. With synthesized system software, this means the updated application may depend on additional system software components. A mechanism should be in place to allow applications *and* system software to evolve in sync. We support application evolution by determining additional VM services required and computing an incremental update that is distributed with the new application. Incremental binary update solutions [22, 12] have been proposed to reduce the cost of sending application and system software binary updates. The modified application is recompiled and rebuilt, and a diff is computed with the original program image. The diffs are injected into the network, and a bootloader applies patches by executing the scripts.

However, diff algorithms can only track structural changes, and ignore application semantic changes. This can result in unnecessarily large diffs caused mainly by *code shifts*, and have adverse impact on the cost of program memory rewriting at the recipient node. When functions in an application change, they may grow or shrink. The resulting code shift requires a large number of pages to be rewritten. Additionally, addresses of subsequent functions change and all references to these functions need to be updated. Thus, even small changes in application structure may require large diff scripts to patch these references, and shift code. To deal with this, we apply our incremental update technique described in [14]. The technique allows us to capture the actual changes, as opposed to structural changes that occur due to dependencies among program elements. We describe the incremental update method further in Section 4.3.

4. IMPLEMENTATION

VM* is a large framework, with components that encompass a broad design space. At the network management level, a base station serves as a repository for application classes and orchestrates the tasks of synthesis, application deployment, and software evolution. Due to space constraints, we focus on the implementations of the core components of VM*. The current implementation is for the Mica family of nodes [5]. These nodes may be interfaced to external sensor boards and include 512K of external flash memory. They contain the Atmel ATMega128 microcontroller [2] with 128K program memory and 4K internal SRAM.

4.1 Application preparation

Java class files contain substantial symbolic information and redundant *constant pool* (CP) [2] entries, which make them large. In fact, only 20% of an average class file is taken up by the method bytecodes [21]. The CP itself occupies 60% of an average class file. Thus, class compaction is a priority. Doing so also reduces the cost of transmitting classes over-the-air for reprogramming. We distinguish between compacted and compressed (or *wire*) formats. Compacted classes are stripped down classes that can be directly understood by the execution engine, while compressed classes require decompression prior to execution. Compression by itself is not helpful because the decompressed class may still be too large for end devices. Compaction is necessary, while compression may be optionally used for further savings in code distribution cost.

Our main compaction strategy is to eliminate redundant string information representing symbolic references from the CP. We encode strings and directly refer to these encodings in place of CP references. This effectively preresolves references, which is normally done at runtime in the JVM. Some application-specific CP entries (e.g., constant values used by ldc instructions) are retained, but shifted out into specialized structures. We also remove class attributes and meta-information normally used by debuggers but not necessary for interpretation. These operations yield 75–80% size reduction for class files (Section 5).

Runtime class representation:

Runtime representations of classes and their instances (objects) in most VM implementations tend to use organized data structures with pointers to internal details such as method tables, etc. We opted for a serialized representation. Doing so eliminates most memory pointers, saving at least 25–30 bytes for typical classes depending on the number of methods, implemented interfaces, etc. Also, classes can be stored in the SRAM or flash (if SRAM space is limited), with minimal changes to the execution engine. Memory switches are difficult with traditional runtime representations because flash and SRAM are accessed very differently. With a serialized format, the only memory pointers to deal with are the class structure's pointers to static field tables. Another advantage is that class and object serialization if required are much simpler because the representation itself is serialized. Appendix A describes the runtime representation for classes. The flat class format, and preresolution of class references required a few changes to some Java bytecodes. Most of these changes are due to the elimination of the CP, and are summarized in Appendix B.

The base station parses and analyzes class files. It also maintains a *registry* of classes, augmented with meta information such as dependencies, and references to native methods. After classes are registered, they are rewritten to conform to our modified bytecode format. During rewriting, targets of some $goto$ instructions need to be fixed, as sections of bytecode can shift due to rewriting. Also, native method *relocations* are created for method invocations through the native interface. These relocations are patched after the VM is built and addresses of native methods are known.

[2]The constant pool is similar to a symbol table, containing strings, class names, constants, etc., referenced by the class.

4.2 Runtime environment

VM*'s runtime environment consists of an interpreter-based execution engine which executes Java bytecode, a lightweight native interface for device-specific functions, and OS support for scheduling and dynamic memory management.

4.2.1 Interpreter implementation

The flexibility afforded by a VM approach comes at the cost of runtime performance overhead. Considerable work has been done on narrowing the performance gap between interpretive schemes and native execution [6]. This is especially important in the WSN domain because performance overheads translate to increased energy requirements. The most effective approaches to mitigate interpretation overhead use either Java to native code compilers [25] or Just-In-Time (JIT) techniques [3]. Java to native code compilation destroys the platform independence that we desire and JIT compilers are more practical on higher end platforms.

In order to preserve platform-independence, we decided to optimize the interpreter by identifying and addressing the primary sources of overhead. Interpreter overhead consists of (i) fetching the next bytecode, (ii) decoding and starting the instruction, and (iii) executing the bytecode implementation. The first two steps constitute the dispatch overhead. When simple bytecodes are executed, the dispatch sequence is comparable in complexity to the bytecode implementations, and incurs considerable overhead. This calls for a more efficient fetch and decode loop. A simple switch based dispatch scheme is straightforward to implement, but it can be inefficient due to range checks enforced by some compilers. An immediate alternative is to use a bytecode table containing addresses of bytecode implementations. Another approach that works well in practice is threaded dispatch [6]. There are several variations of this technique, but the fundamental idea is to incorporate the dispatch operation at the tail of every bytecode implementation instead of returning to a central interpreter loop. Classic threaded dispatch replaces bytecodes with the address of their implementations so the interpreter can directly jump to the implementation without any decoding overhead. Since we use Java bytecode, we cannot directly encode program memory addresses which are two bytes wide on the AVR in the bytestream. Instead, we implement our system in GNU C and approximate threaded dispatch by using labels as values and storing them in an opcode table. This is only an implementation choice — threaded interpretation can be realized without this feature in a few lines of assembly. Because we have to decode the bytecode through a table lookup, we call this scheme *quasi-threading*. Simplified versions of the baseline (non-threaded) and quasi-threaded interpreter are shown in Figure 4.

```
/* NON-THREADED DISPATCH */
while ( 1 )
{
  opcode = fetch_1_byte();
  i = opcode_table[opcode];
  i();
}
```

```
/* QUASI-THREADED DISPATCH */
static void *op_labels = {&&op1, &&op2..};
opcode = fetch_1_byte();
goto *(op_labels[opcode]);
op1:
  <op1 implementation> ...
  opcode = fetch_1_byte();
  goto *op_labels[opcode];
op2:
...
```

Figure 4: Interpreter implementation.

Table 1 summarizes the overheads of the dispatch sequence in various interpreter modes. Classes may be stored in flash or in

SRAM. When stored in flash, they are said to be **ROMized**. SRAM is scarce, but reading bytecodes from SRAM is faster than reading from program memory. The cycle count for the decode/start phase includes the table lookup and instruction start. The non-threaded interpreter uses a function call to start the bytecode execution. Implementing bytecodes using functions ensures maximum benefit from using the fine-grained update model we outline in Section 4.3. The table lookup and indirect call incurs a 18 cycle fixed cost and a variable penalty depending on how many registers are pushed by the bytecode implementation. Most of the fixed cost is due to the table lookup. With the threaded interpreter, the instruction decode and indirect jump to the bytecode implementation costs 13 cycles. The loop count accounts for the return from the bytecode implementation and jump in the main loop of the non-threaded interpreter. With the threaded interpreter, there is no explicit looping since the interpreter loop is effectively unrolled at the tail of each bytecode implementation.

	Non-threaded		Threaded	
	ROMized	SRAM	ROMized	SRAM
Fetch	27	13	27	13
Decode/Start	18	18	13	13
Loop	10	6	-	-

Table 1: Interpreter overheads in clock cycles.

Almost all embedded platforms in wide use for WSN research use RISC processors, but most VMs (including VM*) are stack oriented. Stack machines are popular because they can be implemented with an interpreter more easily. For example, operands do not need to be directly present in the bytestream, as they are implicitly available on the stack. Also, high level languages are easily compiled to stack machine code, and the compiled code tends to be more compact than register oriented code. Unfortunately, this places heavy demands on the register architecture of the underlying hardware. Several researchers have studied the problem of efficiently implementing stack machines in software over register-based hardware. Most of the solutions require architectural support such as superscalar units and deep instruction pipelines which are not available on the devices typical in WSNs. The stack-register impedance mismatch causes several data memory loads and stores because the bytecode, stack, and VM registers are in data space. In some cases, it is helpful to place key VM registers such as the stack pointer, current frame pointer, and instruction pointer in actual hardware registers. However, reserving too many registers will eventually reach a point of diminishing returns due to limited alternatives for the native compiler's register allocation.

Although even carefully crafted interpreters cannot match the performance of native execution, the benefits of the VM approach outweigh the moderate performance penalty that is incurred. We believe the use of threaded dispatch, register mapping, and other optimizations whenever possible, strikes a right balance between performance and platform independence.

4.2.2 Bytecode implementation

We designed the implementation of the JVM instruction set in VM* to be flexible along a number of dimensions. For example, the bytecode fetch is implemented as a macro that is selected during synthesis from a family of macros depending on where classes are stored (flash or SRAM). Non-threaded and threaded versions are also selected during synthesis depending on which interpreter mode is used. A typical implementation is shown in Figure 5 for the $iinc$ (increment local variable by constant) Java bytecode. Bytecode implementations vary in complexity, but are similar in structure.

```
#if QUASI_THREADED
{
instruction_iinc:
#else
void instruction_iinc()
#endif
{
  u1 local_num;
  s4 val;

  local_num = fetch_1_byte();
  val = (s4)stack_frame_getlocal( local_num );
  /* sign extend the const to a byte */
  val += ((s1)(fetch_1_byte()));

  stack_frame_setlocal( local_num, val );
}
#if QUASI_THREADED
opcode = fetch_1_byte();
goto *opcode_labels[opcode];
}
#endif
```

Figure 5: Bytecode implementation.

Interpreter designs prefer jump instructions to function calls to execute bytecode implementations. We use this approach in the quasi-threaded version of the interpreter. However, this increases the cost of extending VM* at runtime if necessary. Using functions to implement bytecodes allows more fine-grained evolution. The tradeoffs are described further in Section 4.3.

4.2.3 Native interface

VMs require access to native functions in order to perform meaningful operations that depend on underlying OS services. VM* provides a lightweight native interface to provide low-level access to device-specific features. Native interfaces are important because some WSN applications can benefit by doing their own specialized hardware resource management. It is also useful for implementing critical sections of computation that need greater execution efficiency than possible with interpreted bytecode. The VM-native layer boundary is a classic tradeoff between efficiency and flexibility. A thin native layer that pushes functionality to the VM, increases portability and may also decrease the static footprint of the implementation. However, performance is worse, which often justifies a larger native layer. Also, having an expressive native interface allows for efficient application-level control over lower layer functions.

VM*'s native interface maps Java types to native types, and exports a set of routines that pass parameters to and from the Java stack. Native method implementations can use these routines to safely exchange data between the VM and native data spaces. Figure 6 contains an example of a native method implementation that uses these exchange routines. The current native interface does not support the ability to access Java references (with the exception of arrays) from the native side. Although we have found the current interface to be more than sufficient for implementing a useful native layer for VM access, we plan to add comprehensive support for VM data space access in the future.

Native functions are statically linked into the VM when it is built. After building the VM, addresses of native methods are read from the VM binary and patched into the method headers in the application classes. Thus, the patching process interacts with our linker, which we describe in Section 4.3. The method header for a native method sets the $code\_length$ field to 0, with the $address$ field set to the actual address of the native implementation. Method invocation instructions check the $code\_length$ field to determine if a native call or virtual method call is to be made.

We have implemented native interfaces for the Mica platform. These include the radio, UART, timers, sensor boards, leds, internal EEPROM and external flash. Synchronous and asynchronous interfaces are provided when relevant. We chose to implement most of the interfaces using static native methods. Using Java interfaces to represent abstract classes that can be implemented by device-specific classes is more elegant, but interface method invocations tend to be inefficient, as interface vtables need to be searched at each invocation for the implementing method.

```
// Java class
package senses.platform.mica2.net;

public class CC1000 {
  static byte state;
  static {
    CC1000.init();
  }
  private static native void init();
  public static native byte sendRadioMsg( byte []data, char sz );
  public static native byte asendRadioMsg( byte []data, char sz );
  public static native byte sendRadioMsgUntilTimeout
                           ( byte []data, char sz, long ms );
  public static native char recvRadioMsg( byte []data );
  public static native char arecvRadioMsg( byte []data );
  public static native char recvRadioMsgUntilTimeout
                           ( byte []data, long ms );
}
```

```
/* Native (C) implementation */
RETURNTYPE_BYTE Java_CC1000_sendRadioMsg() {
  JReference array_ref;
  u1 result, n;
  u1 *buf;

  array_ref = GetReferenceParameter( 1 );
  n = (u1) GetCharParameter( 0 );
  buf = (u1 *) GetArrayContents( array_ref );
  result = send_radio_msg( buf, n);
  ReturnByte( result );
}
```

Figure 6: CC1000 radio native interface.

4.2.4 Operating system support

OS* [19] is a scalable component-based operating system that implements device drivers, memory and resource management, and provides the concurrency framework necessary for scheduling VM tasks and low level event handlers. It currently identifies two kinds of tasks: long running tasks (LRT) and run to completion tasks (RCT). LRTs denote computations that typically span application life time, and can block, yield to other tasks, and be rescheduled by the scheduler. An example of a LRT is the VM interpreter. RCTs denote small amounts of computation that must be performed quickly (typically, event handlers). RCTs do not block, and have higher priority than LRTs. When the scheduler selects a LRT task for execution, it runs until it yields to the scheduler, or until an event occurs. If the LRT yields, the scheduler selects another LRT for execution. If an event occurs, the LRT is preempted and the RCT associated with the event is executed to completion.

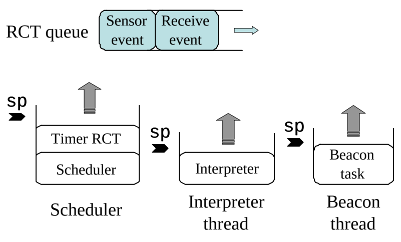


Figure 7: VM* concurrency example: running three threads with outstanding events.

Figure 7 shows a Java application executing in parallel with a beacon thread which transmits radio beacons periodically. This is typical in applications that require dynamic routing tree maintenance as a background service. The interpreter and beacon threads are represented by LRTs and are bound to separate execution contexts. The scheduler runs all RCTs on its own stack, and switches between different LRTs based on task priorities and scheduling policies. The interpreter executes bytecodes until it encounters a blocking call, in which case it yields to another thread, or is interrupted by the hardware. Control is then transferred to the kernel scheduler, which schedules all RCTs for execution before resuming the interpreter thread. In our current implementation for Mica nodes, we allocate 256 bytes for all non-main stacks. The cost of switching between different execution contexts on the Mica platform is about 100 clock cycles. We are currently examining various scheduling policies that may be selected by the synthesis tool based on application and hardware characteristics. Simple applications may use a scheduler optimized for managing a few tasks, while more complex applications will use more advanced policies such as priority-based scheduling.

OS* also contains a two level memory management subsystem for managing SRAM. A low level manager takes care of allocating stacks, neighbor lists, and radio message queues. A VM level manager handles allocation requests for application objects, and implements a mark-sweep garbage collector [27]. The VM's memory manager contains a region-based allocator in which regions consolidate a fixed number of memory objects. This allows sharing of object type information, size, and other fields used by the garbage collector reducing the average object header overhead. The applications considered in this paper were written with the select interface and did not require object allocation. Applications written using the action listener model use dynamically allocated events and event handlers.

4.3 Incremental linking and extensibility

The incremental linker adds tremendous flexibility to our framework, as it provides a fine-grained yet general model for extensibility. While it is primarily intended to allow for seamless evolution of system software with applications, almost any functionality can be incrementally added. The linker is also used to patch native method calls during class registration. The incremental linker is only needed if application evolution is likely, in which case the necessary code distribution and bootloader routines are included in the system software. Otherwise, a standalone VM-application composite binary can be programmed into the node.

Our approach is primarily based on the idea that delta sizes can be reduced if code shifts are reduced. Especially in small, incremental updates (e.g., a function grows by a few statements), code shifts reflect structural changes and not necessarily changes in application semantics. To deal with the complications of code shift when system software needs to change, we modified the linking process to function incrementally. When laying out code in memory, functions are provided with a small amount of *slop* space. If an update causes a function to shrink or grow, it can do so without running into the following function. If an overflow occurs, it is moved to a fresh region with additional slop, without shifting subsequent code. Only references to shifted functions need to be patched. This effectively ensures that the diff process is driven by changes in application semantics. The pages of flash that need to be rewritten are diffed with corresponding pages from the previous image, and a concise delta is generated. This scheme results in substantial savings both in delta size and number of pages that need to be rewritten, over non-incremental approaches (Section 5.6). By making the linker incremental, the program's structural changes are proportional to the semantic changes in the application. As a result, computing a diff between the original program memory image and the updated image results in deltas that are consistent with the extent of actual adaptation.

There are tradeoffs in using the incremental linker when functions are inlined and compiled with optimization. With the threaded interpreter, bytecode implementations are effectively inlined in the interpreter loop, creating a large interpreter function. If several new bytecodes are added due to application changes, the entire interpreter function needs to be shifted. A non-threaded interpreter, on the other hand, is not affected by the addition of new bytecodes — only the individual bytecode implementations need to be added. Even in such cases, using the incremental update model maintains its advantages over pure diff or whole system reprogramming approaches. Details of the linker may be found in [14].

5. EVALUATION

To evaluate VM*, we first studied the relative tradeoffs of the interpreter implementations outlined in Section 4.2.1. The results of these experiments motivated the use of the threaded interpreter with ROMized classes in subsequent experiments. We also wrote a number of applications in VM*, TinyOS and Maté's networking variant (Bombilla), and performed comparisons of their static and dynamic memory footprints, and CPU overheads. These included simple applications such as CntToLeds (CTL), CntToRfm (CTR), RfmToLeds (RTL), and SenseToRfm (STR), similar to the examples provided in the TinyOS distribution. To evaluate a more practical networking application, we implemented a multihop data collection system (*Surge*) using our system. Performance studies were done mainly through simulations in the AVRORA framework [24] with custom monitors to extract information such as bytecode overheads, interpreter instruction issue rate, memory usage, etc. Due to AVRORA's simplified radio model, the Surge application was evaluated with a real deployment. In addition to the basic experiments, our evaluation for Surge included actual deployments in which we measured the packet delivery ratio for various topologies. Finally, overheads in updates were evaluated by performing transformations between application pairs and generating deltas.

5.1 Interpreter performance

We compared the performance of the non-threaded and threaded interpreters, using three synthetic benchmarks. The parameters of interest are average instruction issue rate, and bytecode execution costs. All three tests involved computation in tight loops, and measurements were made for runs lasting 480 seconds. The first test is a loop with simple arithmetic which when compiled, contains only simple bytecodes such as $iinc$ and $bipush$. The second test evaluates the cost of virtual method invocation by performing the same computation in a virtual method invoked from the loop. The third test is a tight loop containing a single invocation of a simple native method, to evaluate the cost of native method invocation.

	Non-threaded	Threaded			
	ROMized	ROMized	%	SRAM	%
I	69994.55	81019.32	15.75	114306.32	63.31
II	38712.34	47585.66	22.92	56992.42	47.22
III	27346.31	30900.75	13.00	31162.05	13.95

Table 2: Interpreter instruction issue rate (instructions per second).

Table 2 shows the instruction issue rates of the two interpreter variants, with classes stored in flash and in SRAM. The percentage improvement reported is with respect to the non-threaded interpreter with ROMized classes. Threaded interpretation yields 15–20 percent improvement in issue rate with ROMized classes, and up to 60 percent improvement with classes in SRAM. The third test does not yield comparable improvement with classes in SRAM because of an issue with the C compiler that forced us to manually push certain registers onto the stack prior to the native method invocation. Although issue rate is much higher with classes in SRAM, SRAM memory is scarce on the ATMega128, and may be a more suitable configuration on platforms such as the Telos. The application comparisons in the following sections were performed using the threaded interpreter with ROMized classes.

The numbers indicate that reduction in instruction issue rate is about 45 percent for virtual method invocation and 60–70 percent for native method invocation. The apparent slow down is caused by two factors. First, virtual method tables record method addresses using 4 byte fields that need to be parsed into 2 byte program memory addresses on the ATMega128. The code length field also needs to be checked, to decide between a native and non-native method invocation. Second, for native methods, several cycles may be spent in the native layer, outside the interpreter loop. Thus, the instruction issue rate will decrease when specialized computation is implemented natively, but the effective CPU utilization of the application will be higher. This also applies to complex bytecode implementations. For example, the threaded interpreter with ROMized classes has an overhead of 40 cycles (Table 1). With a sequence of trivial bytecodes such as $bipush$, the interpreter overhead will be nearly 50 percent, even though instruction issue rate will be higher than when executing complex bytecodes. This suggests that it will be beneficial to identify patterns of computation in an application, and isolate them into complex application-specific bytecodes [20].

Bytecode	$\overline{cycles}$	σ_{cycles}
$bipush$	48.0	0.0
$iadd$	50.0	0.0
$goto$	59.0	3.67
$return$	159.06	6.45
$putfield$	238.10	7.88
$getfield$	311.13	9.11
$invokestatic$	461.18	10.92
$invokevirtual$	546.22	11.97

Table 3: Bytecode overhead in clock cycles.

Table 3 shows the average cost in cycles of selected bytecodes. The non-zero standard deviations are due to branches within bytecode implementations, and can vary across applications. Although bytecodes can be rigorously optimized, performance will be limited by the overheads inherent in the stack oriented JVM architecture. For example, the $iadd$ bytecode costs 50 cycles, while a native implementation of 4 byte signed addition costs less than 10 cycles. Moreover, the JVM specification requires stack entries to be 4 bytes wide, which results in expensive stack operations: most of the values pushed on the stack are 1 or 2 bytes wide, but are promoted to 4 byte values prior to pushing. This also makes the implementation of optimizations such as stack caching difficult because the number of possible stack states becomes very large. It is possible to implement 2 byte stacks, but this requires modifying the Java compiler.

5.2 Application development

The basic applications we considered have very simple operational patterns easily expressed using synchronous native calls. Surge was implemented using the select interface. A snippet from the Surge application's core is shown in Figure 8. Surge is a multihop application in which nodes take photo sensor readings every 2 seconds and send them to a base station through a multihop routing layer. The multihop routing algorithm is similar to the algorithm used in TinyOS, and is implemented natively. The Surge implementation in Maté also sends data to the base station using a *send* primitive, which is implemented in the native multihop routing component. The routing service handles tree maintenance by periodic transmissions of beacons containing neighborhood information. Each node selects a parent based on estimated link qualities of immediate neighbors.

```
SurgePacket sgPkt;
char eList, eVector;
byte sHandle;
sgPkt = new SurgePacket();
evList = Select.setEventId( eList, Events.TIMEOUT | Events.RADIO_RECV );
sHandle = Select.requestSelectHandle();
char val;
Clock.startTimeout( 2048 );
while (true) {
  eVector = Select.select(sHandle, eList);
  if (Select.eventOccurred( eVector, Events.TIMEOUT )) {
    val = PhotoSensor.sense();
    sgPkt.setReading( val );
    Surge.sendPacket( sgPkt );
    Clock.startTimeout( 2048 );
  }
  else if (Select.eventOccurred( eVector, Events.RADIO_RECV)) {
    handleRadioEvent( sgPkt ); // if base, forward to uart
  }
}
```

Figure 8: Surge application snippet.

The multihop routing service includes a beacon thread that periodically sends out beacons for dynamic tree maintenance and a second thread for processing incoming packets. These tasks and the interpreter task are scheduled as LRTs, with context switches occurring when interrupts fire.

5.3 Memory footprint

Figure 9 shows the total size of all the classes in each application. Eliminating redundant CP entries and bytecode rewriting yields savings of 75–80% in class size. Sizes of class files can be reduced even further, if application extraction techniques [23] are used. Since we do not use application extraction, footprints of some applications are similar because they include a common set of system classes (e.g., *Led*, *CC1000*) even if they invoke only a small subset of the methods exported by them.

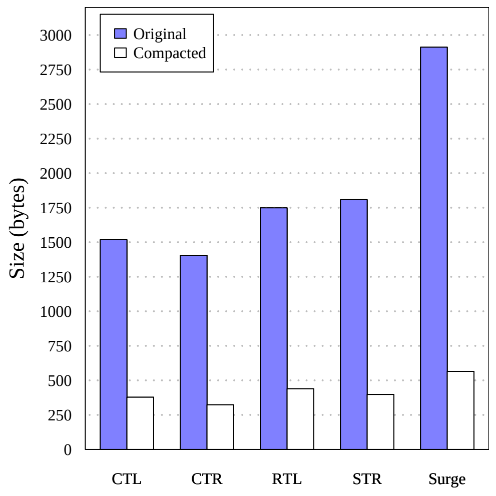


Figure 9: Original and compacted class sizes.

As a result of using software synthesis, only bytecode implementations that are used by the application are included. This is significant because most applications use only a small subset of the JVM instruction set. CTL, CTR, RTL, and STR all use approximately 10% of the instruction set, while Surge uses 14.5% of the instruction set.

Figure 10(a) summarizes the program memory footprints of the applications, and the breakup between the application, VM, and OS. Both TinyOS and Maté benefit from being built using a single generated source file, which allows for better inlining, dead code and variable elimination, and other inter-procedural optimizations. For these experiments, we used the standalone mode of the VM, with the VM binary and application bytecodes flashed in together. Thus, the code distribution and bootloader code are not included in the footprints shown. Miscellaneous code such as standard library functions, and startup code inserted during the final link are included in the OS footprint. By building upon a comprehensive service suite in the system software, class binaries become extremely small. This lends to efficient application distribution, which is critical in the presence of stringent energy and bandwidth constraints. Most embedded processors in WSNs are RISC architectures, characterized by very large code sizes. Thus, the potential energy savings in code distribution are significant.

For comparison with Maté, we used the Bombilla variant which has a flash footprint of 38K. It includes the multihop routing and code distribution components by default, and always contains all bytecode implementations. It also implements a capsule analyzer to determine shared resources used by capsule handlers, for implicit synchronization between contexts. Thus, Maté's approach is to support low-cost application reprogrammability within a domain, at the expense of a one-time deployment of a large VM. A significant domain shift, however, will necessitate deployment of a new VM built from scratch.

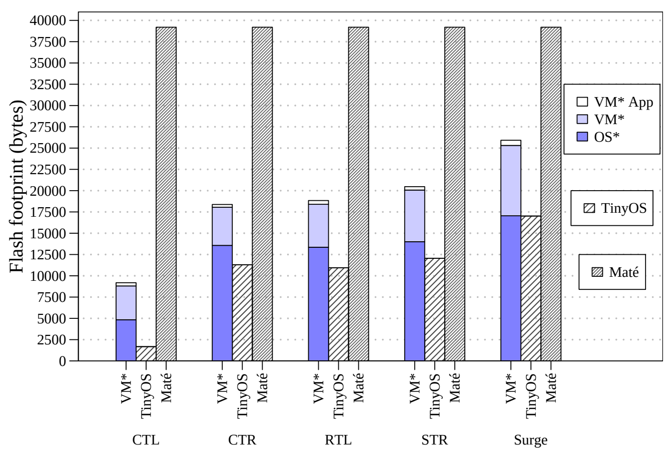
Figure 10(b) gives the SRAM footprints of the applications. It also reports the maximum stack depths and dynamic memory usage observed during execution. For VM*, the Java stack depth is also shown. As mentioned earlier, the JVM requires stack entries to be 4 bytes wide. For example, although the maximum Java stack depth for CTL is 44 bytes, this corresponds to 11 pushes. The larger data section in VM* is mainly due to the threaded interpreter's opcode table, which is between 40 and 70 bytes for these applications. In addition to objects created at runtime, the VM dynamically allocates structures such as the class table and field tables at startup. The applications we considered did not require dynamically allocated objects, except for a Surge packet object created in the Surge application. The OS also uses dynamic memory for stacks, message buffers, execution context state, and neighbor entries as needed within the routing component. Both TinyOS and Maté statically allocate all their variables, which is why the footprints are higher for the Surge application. Maté's application footprint is minimal, due to the specialized instruction set. Application capsules are injected at runtime, and loaded in SRAM for execution.

5.4 Energy overhead

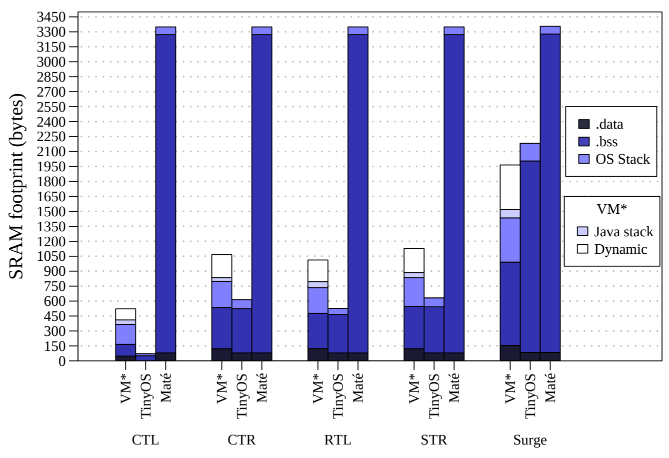
In order to maximize application lifetime, most WSN applications place nodes in a sleep state for most of the time, waking up intermittently to handle events. To satisfy this requirement, the overheads of the system software itself should be low. We measured active times for the test applications over 480 second simulations. The results are shown in Table 4, with corresponding numbers for TinyOS and Maté implementations.

Application	VM*	TinyOS	Maté
CTL	0.23	0.06	6.14
CTR	6.48	6.13	6.43
RTL	6.04	5.65	6.63
STR	5.92	5.78	6.62
Surge	6.09	5.90	6.20

Table 4: CPU activity as percentage of total time in active mode.



(a) Flash memory footprint



(b) SRAM memory footprint

Figure 10: Memory footprints of application and system software.

VM★'s overhead is higher than TinyOS primarily due to the OS scheduler. In TinyOS, the FIFO scheduler schedules tasks that run to completion without any context switching required, except when events occur, in which case short running event handlers are executed. In addition to the scheduler overhead, the interpreter and bytecode overheads contribute to the active time. However, these overheads are not as severe as the scheduler, especially in applications that sleep most of the time. VM★ compares favorably with Maté. Maté's overheads are mainly due to scheduling (each bytecode is executed as a TinyOS task), and the code update mechanism which runs in the background.

5.5 Surge evaluation

We implemented Surge in VM★ and compared the average packet delivery ratio with TinyOS and Maté. The main objective was to determine if the additional overheads in system software resulted in any noticeable degradation in the communication layer. A collector at the base station processed packets received from the network during 30 minute runs to compute packet delivery ratio.

All Surge and routing service parameters were identical in the three systems. The routing component transmits a beacon every 20 seconds, with parent selection performed every 5 beacons. Sensor readings are transmitted every 2 seconds. We repeated the experiment with a 5-hop chain, and 2x2, 3x3, and 4x4 grids, with the base station placed at a corner of the mesh. Packet delivery ratio in any topology is affected by a number of factors. During a run, a single node may become a transit point for multiple disjoint paths and choke descendants in those paths. In grid topologies, nodes change their parent relatively frequently, especially in the presence of a number of candidate neighbors. For fair comparison, we avoided these effects by setting up a fixed routing tree. Nodes continue to send beacons and incur the overhead of parent selection, but the parent selection function is forced to set the parent to a fixed node. Radio transmission power was at a low setting to reduce congestion.

Table 5 shows the packet delivery ratios for the topologies considered. In the sparser topologies, in-degrees of nodes are usually low resulting in higher delivery ratios. The 4x4 deployment is affected by higher in-degrees, larger network diameter, and collisions in dense neighborhoods. The results are comparable on all systems.

5.6 Incremental linking

Topology	VM★	TinyOS	Maté
2 x 2	97.60	99.15	98.60
3 x 3	92.44	92.32	90.77
4 x 4	78.10	79.32	77.24
5 hop chain	95.66	98.88	93.47

Table 5: Surge average packet delivery ratio.

The incremental linking technique was designed to reduce the effects of code shift. Most updates in deployed WSN applications are likely to be incremental in nature. In some cases, complete retasking may be necessary. Even in these extreme scenarios, significant portions of the system software such as device drivers may remain the same.

To evaluate the update mechanism's performance, we incrementally transformed CTL to CTR, and then CTR to Surge. CTL to CTR requires the addition of communication device drivers at the system software level. CTR to Surge requires the addition of the multihop routing component. Even in application upgrades, the system software delta tends to be much larger than the application delta.

Table 6 shows the two update scenarios we considered. In both experiments, the non-threaded version of the interpreter was used. In Table 7, the $copy$, run, and add instructions are diff-script opcodes. The $copy$'s copy data from the existing image, while run and add opcodes introduce new data. The data column indicates the total size of the added data, opcodes, and address tables in the diff encoding. The percent value of the new binary size is the measure of improvement over whole system reprogramming.

Application pair	Old size (bytes)	New size (bytes)
CTL→CTR	16795	29239
CTR→Surge	29239	34834

Table 6: Incremental update scenarios.

Application pair	*copy*	*run*	*add*	Data (bytes)	% Binary
	Pure diff				
CTL → CTR	1125	6	671	13233	45.26
CTR → Surge	1426	5	812	15280	43.87
	Incremental linking				
CTL → CTR	884	13	495	9070	31.02
CTR → Surge	734	15	428	8161	23.43

Table 7: Delta footprint with pure diff and incremental linking.

Both diff and incremental linking approaches perform significantly better than whole system reprogramming. Delta sizes for incremental linking are 30–45 percent smaller than the pure diff delta. For updates that are more incremental in nature, such as modifying a few functions as opposed to introducing new ones in an application upgrade, the benefits of the incremental approach will be more substantial.

6. CONCLUSION

Developing applications in mainstream distributed systems is difficult. WSNs pose additional challenges due to diverse platforms, large scale, and resource limitations. Meeting the impressive potential of WSN applications forecast by researchers in various disciplines is only possible with the right system software. We have surveyed the critical challenges in ensuring long-term viability of WSN applications, and described a framework which implements a VM approach to prepare for the heterogeneity likely in realistic deployments in the near future. Through fine-grained software synthesis of a virtualized architecture, an optimized execution engine, and a flexible model for extensibility, the framework achieves a balance between a number of competing concerns. Although it implements a VM approach, and is tailored around the JVM, we believe other system software approaches can benefit from using software synthesis in tandem with incremental linking for application and system software co-evolution.

6.1 Future work

VM* is ongoing work, and we are in the process of implementing the framework on various platforms. We are also exploring the VM design space to understand what programming and computation models are suitable for WSN applications. Our experiences with VM* suggest that an efficient computation model and well designed OS support are prerequisites for the viability of the approach. In particular, we will evaluate options such as register oriented VMs that may yield better performance on the RISC microcontrollers typical in WSNs. Using register oriented VMs that are close to the native architecture may also enable lightweight JIT compilers. We also plan to explore various instruction sets tailored to specific application domains. We anticipate that further studies will lead to more powerful programming models, better system software, and hints for architectural choices in sensor node design.

7. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and our shepherd Matt Welsh for valuable feedback. We also thank our collaborators at UC Davis: Earl Barr, Eric Bergstrom, Paul Hans, Jeyasankar Kottalam, Ingwar Wirjawan, and Jeff Wu. Special thanks to Ben Titzer and the UCLA Compilers Group for providing the AVRORA toolset, Kiem Vo (AT&T Research, NJ) for useful advice on diff algorithms, and Ram Kumar (UCLA) for suggestions on the experimental evaluation.

8. 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 ACM International Workshop on Wireless Sensor Networks and Applications*, pages 50–59, San Diego, CA, September 2003. ACM Press.

[2] Atmel Corporation. *ATMega128 Datasheet*.

[3] J. Aycock. A Brief History of Just-In-Time. *ACM Computing Surveys*, 35(2):97–113, 2003.

[4] G. Bracha, N. Cohen, C. Kemper, M. Odersky, D. Stoutamire, K. Thorup, and P. Wadler. Adding generics to the Java programming language. Java Community Process JSR-000014, September 2004.

[5] Crossbow Technology Inc. *Mica Motes*. `http://www.xbow.com`.

[6] M. Ertl and D. Gregg. The Behavior of Efficient Virtual Machine Interpreters on Modern Architectures. In *Proceedings of the European Conference on Parallel Computing*, volume 2150 of *Lecture Notes in Computer Science*, pages 403–412, Manchester, UK, August 2001. Springer Verlag.

[7] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 1–11, San Diego, CA, 2003. ACM Press.

[8] C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A Dynamic Operating System for Sensor Nodes. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services*, pages 163–176, Seattle, WA, June 2005. ACM Press.

[9] J. Hill, M. Horton, R. Kling, and L. Krishnamurthy. The Platforms Enabling Wireless Sensor Networks. *Communications of the ACM*, 47(6):41–46, 2004.

[10] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System Architecture Directions for Networked Sensors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, Cambridge, MA, 2000. ACM Press.

[11] J. Hui and D. Culler. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In *Proceedings of the International Conference on Embedded Networked Sensor Systems*, pages 81–94, Baltimore, MD, November 2004. ACM Press.

[12] J. Jong and D. Culler. Incremental Network Programming for Wireless Sensors. In *Proceedings of the International Conference on Sensor and Ad Hoc Communications and Networks*, Santa Clara, California, October 2004.

[13] J.Wu and R. Pandey. BOTS: A Constraint-Based Component System for Synthesizing Scalable Software Systems. Technical Report CSE-2005-18, University of California, Davis, August 2005.

[14] J. Koshy and R. Pandey. Remote Incremental Linking for Energy-Efficient Reprogramming of Sensor Networks. In *Proceedings of the European Workshop on Sensor Networks*, pages 354–365, Istanbul, Turkey, January 2005.

[15] P. Levis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *Proceedings of the International Conference on Architectural Support for Programming*

Languages and Operating Systems, pages 85–95, San Jose, CA, 2002. ACM Press.

[16] P. Levis and D. Culler. Active Sensor Networks. In *Proceedings of the Symposium on Networked Systems Design and Implementation*, Boston, MA, May 2005.

[17] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, April 1999.

[18] J. Mulder, S. Dulman, L. van Hoesel, and P. Havinga. PEEROS — System Software for Wireless Sensor Networks. Preprint, August 2003.

[19] R. Pandey, J. Kottalam, Y. Ramin, I. Wirjawan, and J. Koshy. OS*: A Scalable Component-Based Operating System for Sensor Networks *(in preparation)*, 2005.

[20] T. Proebsting. Optimizing an ANSI C interpreter with superoperators. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 322–332, San Francisco, CA, 1995. ACM Press.

[21] W. Pugh. Compressing Java Class Files. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 247–258, Atlanta, GA, May 1999. ACM Press.

[22] N. Reijers and K. Langendoen. Efficient Code Distribution in Wireless Sensor Networks. In *Proceedings of the ACM International Conference on Wireless Sensor Networks and Applications*, pages 60–67, San Diego, CA, 2003. ACM Press.

[23] F. Tip, P. Sweeney, C. Laffra, A. Eisma, and D. Streeter. Practical Extraction Techniques for Java. *ACM Transactions on Programming Languages and Systems*, 24(6):625–666, 2002.

[24] 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.

[25] A. Varma and S. Bhattacharyya. Java-through-C Compilation: An Enabling Technology for Java in Embedded Systems. In *Proceedings of Design Automation and Test in Europe*, pages 161–167, Paris, France, February 2004. IEEE Computer Society.

[26] M. Weiser. Some Computer Science Issues in Ubiquitous Computing. *Communications of the ACM*, July 1993.

[27] P. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of the International Workshop on Memory Management*, pages 1–42, London, UK, 1992. Springer-Verlag.

APPENDIX

A. CLASS FORMAT

Method invocation involves obtaining a pointer to the method header from the vtable. From the header, the code length, number of arguments, local variables, and method address itself are obtained. A subclass includes the vtable of superclasses so that invocations of methods implemented in superclasses can be done without having to search through the class hierarchy. Native methods have their $code\_length$ field set to 0 so that an indirect call can be made to the function at the address specified in the $code$ area. The $ivtable$ is essentially a mapping from interface methods to their implementations. Invoking an interface method involves searching for the interface in the $ivtable$ and obtaining the implementing method's index in the $vtable$. The instance (or object) format is simply a flattened instance field table preceded by an object header which includes information required by the garbage collector and meta information about the object's class type.

FIELD	SIZE	DESCRIPTION
$class\_address$	4	class address (flash/SRAM).
$class\_size$	2	class size (bytes).
$class\_id$	2	encoding assigned during registration.
$super\_id$	2	class ID of the super class.
$n\_interfaces$	1	number of interfaces implemented.
$n\_ventries$	1	number of methods in vtable.
sf_table_length	2	length of static field table (bytes).
if_table_length	2	length of instance field table (bytes).
$main\_id$	1	vtable index of main method.
$clinit$	1	vtable index of static initializer.
Interface list		
. . .		
$interfaceID$	2	encoding assigned to the implemented interface during registration.
$n\_ventries$	1	number of methods implemented for this interface.
. . .		
Vtable		
. . .		
$codeaddress$	4	address of method header.
. . .		
Interface vtable(s)		
. . .		
Interface i		
$\rightarrow m1\ index$	1	vtable index of first implemented method.
$\rightarrow m2\ index$	1	vtable index of second implemented method.
. . .		
. . .		
Method section(s)		
. . .		
Method header		
$\rightarrow code\_length$	2	method code length in bytes (0 if native).
$\rightarrow n\_args$	1	number of arguments to method.
$\rightarrow n\_locals$	1	number of local variables in method.
$\rightarrow code$	-	bytecode for method, or native address if method is native.
. . .		

B. BYTECODE EXTENSIONS

Sizes of bytecode operands if any, are indicated in parentheses. Most of the changes are due to the elimination of the constant pool (CP). For example, in the JVM, the two byte argument to the $checkcast$ instruction is an index into the CP which contains the referenced class. Since we preresolve references, this index is replaced by the runtime encoding of the referenced class.

BYTECODE	DESCRIPTION
$ldc\{descriptor(2)\}$	$descriptor$[15:14] indicates size of constant, [13:0] encodes offset into specialized constant table.
$checkcast, instanceof, new$ $\{cid(2)\}$	cid is the 2 byte encoding assigned to the class during registration.
$getfield, putfield$ $getstatic, putstatic$ $\{cid(2)\}\{fid(2)\}$	cid and fid are the 2 byte encodings assigned during registration.
$invokeinterface$ $invokespecial$ $invokestatic\{cid(2)\}\{mid(1)\}$	cid and mid are the 2 and 1 byte encodings assigned during registration. Index mid of the vtable points to the header of the method to be invoked.
$invokevirtual$ $\{o(1)\}\{mid(1)\}$	Object is retrieved at offset o from top of stack. Index mid of the object's vtable points to the header of the method to be invoked.
$lookupswitch, tableswitch$	Same as JVM specification, with padding removed.