Bombilla: A Tiny Virtual Machine for TinyOS

Philip Levis
pal@cs.berkeley.edu

Version 1.0
September 25, 2002
Figure 1: Bombilla Architecture and Execution Model: Capsules, Contexts, and Stacks

| basic | 00111111 | i = instruction |
| m-class | 01011xxx | i = instruction, x = argument |
| v-class | 011xxxxx | i = instruction, x = argument |
| j-class | 1011xxxxx | i = instruction, x = argument |
| x-class | 11xxxxxx | i = instruction, x = instruction |

Figure 2: Bombilla Instruction Formats

1 Introduction

Bombilla is a tiny bytecode interpreter that runs on top of TinyOS. Because it presents a high-level communication-centric interface, Bombilla’s programs are very short; combined with a safe execution environment, this makes mote reprogramming rapid and error-free. Programs can self-propagate through a network; this makes reprogramming mostly autonomous, as once a self-propagating capsule is introduced, it will eventually install itself over the entire network.

Bombilla has multiple execution contexts; each runs in response to an event, and they can interleave at instruction granularity. Bombilla prevents race conditions by implicitly locking all shared state that is used; as every resource is statically named and Bombilla programs are short, the set of required locks can be quickly determined with a full program analysis. Program writers can explicitly yield or release locks to improve parallelism.

One goal of Bombilla is to provide a programming interface to motes that is much simpler than TinyOS; a sense-and-send program can be as few as six instructions in Bombilla, and Bombilla’s error detection mechanisms can help novice programmers find the bugs in their programs.

2 System Overview

Bombilla is a set of TinyOS components that sit on top of several system components, including sensors, the network stack, and non-volatile storage (the “logger”). Code is broken in capsules of 24 instructions, each of which is a single byte long; larger programs can be composed of multiple capsules. In addition to bytecodes, capsules contain identifying and version information. Bombilla has two stacks: an operand stack and a return address stack. Most instructions operate solely on the operand stack, but a few instructions control program flow and several have embedded operands. There are three execution contexts that can run concurrently at instruction granularity. Bombilla provides both a built-in ad-hoc routing algorithm (the send instruction) as well as mechanisms for writing new ones (the sendr instruction).

1Support for the logger is currently not implemented, due to the pending addition of a new logger interface in TinyOS.
3 Architecture, Instruction Set, and Data Types

Bombilla instructions hide the asynchrony (and oft-resulting race conditions) of TinyOS programming. For example, when the `send` instruction is issued, Bombilla calls a command in the ad-hoc routing component to send a packet. Bombilla suspends the context until a message send complete event is received, at which point it resumes execution. By doing this, Bombilla does not need to manage message buffers – the capsule will not resume until the network component is done with the buffer. Similarly, when the `sense` instruction is issued, Bombilla requests data from the sensor TinyOS component and suspends the context until the component returns data with an event. This synchronous model makes application level programming much simpler and far less prone to bugs than dealing with asynchronous event notifications. Additionally, Bombilla efficiently uses the resources provided by TinyOS; during a split-phase operation, Bombilla does nothing on behalf of the calling context, allowing TinyOS to put the CPU to sleep or use it freely.

Bombilla is a stack-based architecture. This allows a concise instruction set; most instructions do not have to specify operands, as they exist on the operand stack. There are five classes of Bombilla instructions: basic, m-class, j-class, v-class, and x-class. Figure 2 shows the instruction formats for each class. Basic instructions include such operations as arithmetic, halting, and activating the LEDs on a mote. m-class instructions access message headers; they can only be executed within the message send and receive contexts. v-class instructions access the 16 word Bombilla heap. j-class instructions are the two jump instructions, for loops and conditionals. The one x-class instructions is `pushc` (push constant). All instruction classes except basic have embedded operands.

Bombilla’s three principal execution contexts, illustrated in Figure 1, correspond to three events: clock timers, message receptions and message send requests. Inheriting from languages such as FORTH, each context has two stacks, an operand stack and a return address stack. The former is used for all instructions handling data, while the latter is used for subroutine calls. The operand stack has a maximum depth of 16 while the call stack has a maximum depth of 8. We have found this more than adequate for programs we have written.

There is an additional context, the “once” context. Unlike other contexts, which run their capsules many times, this context only runs its capsule once, when it is installed; this allows a user to initialize state, adjust constants, or perform other operations that only need a single execution.

There are three operands types: values, sensor readings, and buffers. Some instructions can only operate on certain types. For example, the `putled` instruction expects a value on the top of the operand stack. However, many instructions are polymorphic. For example, the `add` instruction can be used to add any combination of the types, with different results. Adding buffers results in appending the data in the second operand onto the first operand. Adding a value to a message appends the value to the message data payload. Sensor readings can be turned into values with the `cast` instruction.

Sensor readings are typed, and cannot be modified. For example, in order to take an average over a set of readings, each reading must be first be converted to a value; these values can then be averaged. Many instructions (e.g. `inv`) automatically cast sensor readings to values. This ensures that a sensor reading variable has some meaning; otherwise, it could express some arbitrary quantity. Adding two sensor readings of the same type (e.g. magnetometer X-axis) produces a value, and adding two sensor readings of different values is an error.

Buffers are also typed, and can hold up to ten values. A buffer can only hold elements of one type, whether they be values or a certain sensor type. An empty buffer has no type; the first element added will set the type of the buffer. Buffers have several access instructions, including `bhead` (copy of the first element of the buffer onto the operand stack), `byank` (pull the n-th element out of the buffer and push it into the operand stack), and `bsorta` (sort the elements in ascending order.

There is a 16 word heap shared among the context. It can be accessed with the `setvar` and `getvar` instructions, which have a 4-bit embedded operand.. This allows the separate contexts to communicate shared state (e.g. sensor readings).

3.1 Capsules and Execution

Bombilla programs are broken up into capsules of up to 24 instructions. This limit allows a capsule to fit into a single TinyOS packet. By making capsule reception atomic, Bombilla does not need to buffer
getvar 0  # Get heap variable 0
pushc 1  # Push one onto operand stack
add  # Add the one to the stored counter
copy  # Copy the new counter value
setvar 0  # Set heap variable 0
pushc 7  # Take bottom three bits of counter
and  # Set the LEDs to these three bits
halt

Figure 3: Bombilla cnt_to_leds – Shows the bottom 3 bits of a counter on mote LEDs

pushc 1  # Push one on the operand stack
sense  # Read sensor 1 (light)
copy  # Copy the sensor reading
getvar 0  # Get previous sent reading
inv  # Invert previous reading
add  # Current - previous sent value
copy  # 2 copies of difference on top of stack
pushc 32  #
gt  # Is current 32 greater than previous?
sup  # Swap result with copy
pushc 32  #
inv  #
lc  # Is current 32 less than previous?
or  # Either 32> or 32<
jumps 15  # Jump to send
halt

copy  # Copy new value
setvar 0  # Set current
bpush0  # Push buffer 0 onto stack
bclear  # Clear its contents
add  # Add current reading to buffer
send  # Send buffer
halt

Figure 4: Bombilla Program to Read Light Data and Send a Packet on Reading Change

partial capsules, which conserves RAM. Every code capsule includes type and version information. Bombilla defines four types of code capsules: message send capsules, message receive capsules, timer capsules, and subroutine capsules. Subroutine capsules allow programs to be more complex than what fits in a single capsule. Applications invoke and return from subroutines using the call and return instructions. There are names for up to $2^{15}$ subroutines; to keep Bombilla’s RAM requirements small, its current implementation has only four.

Bombilla begins execution in response to an event – a timer going off, a packet being received, or a packet being sent. Each of these events has a capsule and an execution context. Control jumps to the first instruction of the capsule and executes until it reaches the halt instruction. These three contexts can run concurrently. Each instruction is executed as a TinyOS task, which allows execution interleaving at an instruction granularity. Additionally, underlying TinyOS components can operate concurrently with Bombilla instruction processing. When a subroutine is called, the return address (capsule, instruction number) is pushed onto a return address stack and control jumps to the first instruction of the subroutine. When a subroutine returns, it pops an address off the return stack and jumps to the appropriate instruction.

The packet receive and clock capsules execute in response to external events; in contrast, the packet send capsule executes in response to the sendr instruction. As sendr will probably execute a number of Bombilla instructions in addition to sending a packet, it can be a lengthy operation. Therefore, when sendr is issued, Bombilla copies the message buffer onto the send context operand stack and schedules the send context to run. Once the message has been copied, the calling context can resume execution. The send context executes concurrently to the calling context, preparing a packet and later sending it. This frees up the calling context to handle subsequent events – in the case of the receive context, this is very important.

The constrained addressing modes of Bombilla instructions ensure a context cannot access the state
of a separate context. Every push and pop on the operand and return value stack has bound checks to prevent overrun and underrun. As there is only a single shared variable, heap addressing is not a problem. Unrecognized instructions result in simple no-ops. All bounds are always checked – the only way two contexts can share state is through \texttt{gets} and \texttt{sets}. Nefarious capsules can at worst clog a network with packets – even in this case, a newer capsule will inevitably be heard. By providing such a constrained execution environment and providing high-level abstractions to services such as the network layer, Bombilla ensures that it is resilient to buggy or malicious capsules.

### 3.2 Simple Programs

The Bombilla program in Figure 3 maintains a counter that increments on each clock tick. The bottom three bits of the counter are displayed on the three mote LEDs. The counter is kept as a value which persists at the top of the stack across invocations. This program could alternatively been implemented by using \texttt{gets} and \texttt{sets} to modify the shared variable. This code recreates one of the simplest TinyOS applications, \texttt{cnt\_to\_leds}, implemented in seven bytes.

The Bombilla program in Figure 4 reads the light sensor on every clock tick. If the sensor value differs from the last sent value by more than a given amount (32 in this example), the program sends the data using Bombilla’s built-in ad-hoc routing system. This program is 24 bytes long, fitting in a single capsule.

### 3.3 Capsule Injector

A tool is included in the TinyOS release to aid in the writing of Bombilla programs: \texttt{net.tinyos.vm.asm.CapsuleInjector}. \texttt{CapsuleInjector} provides an interface for writing assembly programs and sending them to a mote connected to a PC; if the capsule is marked self-forwarding, it will then start propagating into the network.

One must set the destination mote ID of the capsule packet (important when using a GenericBase) and the version number of the capsule. Version numbers are explained in Section 5; Bombilla only installs a capsule if its version number is higher than the one it currently has.

If the program has an error (e.g. a v-class instruction without a specified operand), \texttt{CapsuleInjector} does not send out a packet.

### 4 Synchronization

Bombilla interleaves the execution of its contexts at instruction granularity. The presence of a 16-word shared heap means that if different contexts communicate or share variables (e.g. an aggregated sensor reading), race conditions can easily occur. As the program running on a mote is the combination of possibly forwarding capsules, applications can go through transient states of partial installation, making programmer efforts (e.g. a spin loop) ineffectual.

Bombilla therefore uses an implicit locking scheme, so that programmers are assured that there will be no race conditions in their programs. Experienced programmers can relax the locking requirements to improve parallelism.

![Figure 5: CapsuleInjector GUI](image.png)
4.1 Model

In this section, we define the synchronization model, which is based on five abstractions: handlers, invocations, resources, scheduling points and sequences. We describe how we discover the resources used by an invocation, and how invocation communicate their resource requirements to the runtime system.

A handler is a function that is executed in response to some event. An invocation represents a particular execution of a handler in response to an event. At any time, invocations are in one of four states: waiting (for resources), suspended (waiting for an operation to complete), ready (can execute), running (executing). We say that an invocation that is ready or running is active.

A resource is a shared piece of state that a handler requires access to – examples are a variable, a disk arm, or a sensor. Resources can only be held by invocations.

A handler contains a number of scheduling points at which it can be suspended and gain or lose resources (and resources cannot be acquired anywhere but at scheduling points). Scheduling points are the handler’s entry and exit points, and some subset of its operations which we call scheduled operations. An invocation goes through two states during execution of a scheduled operation: first, the invocation is suspended awaiting the completion of the operation; second, the invocation is waiting for the resources it wishes to gain to become available. Either of these two phases may be trivial: a yield operation completes immediately but may wait for some resources, a message send does not complete immediately but, if it is not waiting for any resources, will not wait. A sequence is any code path between two scheduling points which does not include another scheduling point.

The model for resource acquisition and release is as follows: before an invocation can start execution, it must acquire all resources it will need during its lifetime. At each subsequent scheduling point, the invocation can release a set \( R \) of resources before suspending execution, and acquire a set \( A \) of resources before resuming. To prevent deadlock, we require \( A \subseteq R \) (we prove below that this condition is sufficient for building deadlock-free schedulers). Finally, when an invocation exits it must release all held resources. Note that we do not guarantee any atomicity between two invocations of the same handler.

To preserve safety, the static analysis of a handler’s resource usage and the runtime system must guarantee that an invocation holds all resources at the time(s) at which they are accessed and that the intersection of the resources held by any two invocations is empty. We restrict our invocation model to considering a static number of resources, and require operations to explicitly name the resources they use so that we can easily analyse handlers at compile (or load) time. Resource discovery must be conservative to preserve correctness.

4.2 Bombilla Implementation

We implemented this synchronization model in Bombadillo. Each Bombadillo context is an invocation, and capsules are implicitly broken up into sequences. Bombadillo maintains two queues of invocations: ready and waiting. Whenever Bombadillo installs a new capsule, it performs a static full-program analysis to generate the acquire sets of its invocation start points. Without requiring any annotation from a programmer, Bombadillo runs invocations atomically while allowing parallelism. Programmers can improve the degree of parallelism by yielding resources at scheduling points.

Bombadillo invocations are broken into sequences by scheduling point instructions. When a context executes one of these instructions, the Bombadillo runtime examines the current release set of the issuing context and releases the locks on the indicated resources. Bombadillo then checks the waiting queue to see if any contexts have been made runnable by the release of these locks. When the scheduling point instruction completes, Bombadillo checks the acquire set of the context to see if it can be made active; if so, the context acquires its locks and Bombadillo places it on the ready queue. If the context cannot be made active, Bombadillo places it on the waiting queue.

Bombadillo defines its scheduling point instructions to be those that trigger split-phase operations in TinyOS. This includes acquiring sensor data (sense), sending packets (send, sendr, uart), and accessing
non-volatile flash memory storage \( \log r, \log w, \log w l \). Additionally, there is a \texttt{yield} instruction, which is effectively a split-phase operation that immediately completes. Locks are added to a context’s release set with the \texttt{unlock} instruction – by default, the set is empty. \texttt{unlock} also adds the lock to the context’s acquire set. For a lock to be released, but not re-acquired, a context much use the \texttt{punlock} (unlock permanent) instruction. The \texttt{unlock} and \texttt{punlock} instructions affect individual locks, enumerated by an operand; the \texttt{unlockb} and \texttt{punlockb} instructions use the operand as a bitmask for locks to be released. Locks are not released until a scheduling-point instruction is executed. Figure 6 contains two sample Bombadillo instruction sequences that demonstrate resource unlocking.

Release and acquire sets are atomically handled by Bombadillo. A context does not acquire any locks in its acquire set unless it can acquire all of them, and acquires them atomically. Similarly, release sets are released atomically. This, combined with monotonically decreasing lock sets, ensures the system is deadlock free.

5 Viral Programming

Bombilla code capsules can be marked “forwarding.” The code forwarding system triggers roughly every 700 milliseconds; it takes a rough estimate of the network’s business, and probabilistically decides whether it will forward a capsule. Capsules are forwarded more often in a quiet network than in a busy one. If Bombilla decides to forward a capsule, it chooses one at random and checks if it the forwarding bit is set; if so, it transmits the capsule on an AM broadcast.

Every capsule has a version number. When Bombilla hears a capsule broadcast, it checks if the capsule is newer than the one currently installed; if so, Bombilla halts execution of that context and installs the new capsule.

Our first implementation of the VM had an explicit capsule forwarding system (the \texttt{forw} instruction); experimental results showed this to be a terrible idea, as programs could very easily saturate the network unknowingly. We therefore adopted this simple probabilistic model. It is by no means perfect; for example, even if every mote in the network is running the same capsule, they will continue to forward it indefinitely. We are currently exploring more efficient code propagation mechanisms.

6 Error State

Bombilla has an error state, which can help users debug their programs. If a program triggers an error (for example, by trying to add incompatible sensor readings, or by overflowing the operand stack), Bombilla halts execution on all contexts. Then, every second, it blinks all of the LEDs and sends a packet containing debugging information over the UART. The packet contains the offending context identifier, the capsule it was executing, the instruction that caused the error, and the error code. Error codes can be found in \texttt{tos/lib/Bombilla.h}. They are:

```c
typedef enum {
    BOMB_ERROR_TRIGGERED = 0,
    BOMB_ERROR_INVALID_RUNNABLE = 1,
    BOMB_ERROR_STACK_OVERFLOW = 2,
    BOMB_ERROR_STACK_UNDERFLOW = 3,
    BOMB_ERROR_BUFFER_OVERFLOW = 4,
    BOMB_ERROR_BUFFER_UNDERFLOW = 5,
    BOMB_ERROR_INDEX_OUT_OF_BOUNDS = 6,
    BOMB_ERROR_INSTRUCTION_RUNOFF = 7,
    BOMB_ERROR_LOCK_INVALID = 8,
    BOMB_ERROR_LOCK_STEAL = 9,
    BOMB_ERROR_UNLOCK_INVALID = 10,
    BOMB_ERROR_QUEUE_ENQUEUE = 11,
    BOMB_ERROR_QUEUE_DEQUEUE = 12,
    BOMB_ERROR_QUEUE_REMOVE = 13,
    BOMB_ERROR_QUEUE_INVALID = 14,
    BOMB_ERROR_RSTACK_OVERFLOW = 15,
    BOMB_ERROR_RSTACK_UNDERFLOW = 16,
} bomb_error;
```
BOMB_ERROR_INVALID_ACCESS = 17,
BOMB_ERROR_TYPE_CHECK = 18,
BOMB_ERROR_INVALID_TYPE = 19,
BOMB_ERROR_INVALID_LOCK = 20,
BOMB_ERROR_INVALID_INSTRUCTION = 21
} BombillaErrorCode;

The Bombilla error packet has the following payload:

typedef struct BombillaErrorMsg {
  uint8_t context;
  uint8_t reason;
  uint8_t capsule;
  uint8_t instruction;
} BombillaErrorMsg;