NXTalk

Dynamic Object-Oriented Programming in a Constrained Environment

Martin Beck   Michael Haupt   Robert Hirschfeld

Hasso Plattner Institute, University of Potsdam, Germany

{martin.beck,michael.haupt,hirschfeld}@hpi.uni-potsdam.de

Abstract

Dynamic programming languages offer high expressiveness and flexibility, leading to improved programmer productivity. Still, making dynamic programming languages available for embedded systems is challenging because such environments are often constrained in terms of memory or computational power. For this, it is necessary to reduce the size of language implementations (virtual machines, VMs) while at the same time retaining good performance and robustness. Automatic memory management deserves special attention because its performance and space overhead have noticeable impact on the overall usability of the system. In this paper, we present NXTalk, a VM and programming environment for the dynamic programming language Smalltalk, making high-level object-oriented programming available on Lego Mindstorms NXT robots. We describe its VM implementation addressing the aforementioned challenges, and evaluate its size and performance characteristics.

Categories and Subject Descriptors C.3 [*Special-Purpose and Application-Based Systems*]: Real-time and embedded systems; D.3.4 [*Programming Languages*]: Processors—Interpreters; D.4.7 [*Operating Systems*]: Organization and Design—Real-time systems and embedded systems

General Terms Design, performance

Keywords Embedded system, resource-constrained device, dynamic programming language, virtual machine, Smalltalk, Lego Mindstorms NXT

1. Introduction

Embedded systems are mostly programmed using static programming languages. However, dynamic programming languages have long been applied successfully in this domain [20]. For example, the Smalltalk programming language [10] was used to develop software for Tektronix oscilloscopes [7], and in chip manufacturing by Texas Instruments [20]. More recently, Smalltalk has been used in, e. g., digital loudspeakers [2]; and Python has made and is making appearances in mobile devices, e. g., Nokia Series 60[1] or the Android[2] platform.

Special challenges, but also benefits, result when dynamic languages are applied in the embedded systems domain. One of the most crucial challenges is to make the high-level abstractions that dynamic languages typically bring about actually *work*. Not only are memory and other resources often severely restricted, but there is also a permanent tradeoff between code size and performance. One has to admit, though, that the benefits are strong in their own regard. Automated object layout, memory management and other abstractions make programming and debugging for embedded system environments much more convenient, which leads to significantly shorter turn-around and product accomplishment times [20].

The Lego Mindstorms NXT[3] is a low-cost and easily accessible platform for building robots. It is constrained especially in its memory equipment and represents an embedded system platform with interesting characteristics. Its system architecture is well documented[4] and invites beginners as well as more experienced people to use it to work with an embedded system. It is thus particularly well suited for teaching both robotics and embedded systems programming.

This paper makes the following contributions:

- We introduce the NXTalk[5] project, which brings the Smalltalk programming language to the NXT platform. NXTalk unifies embedded systems programming at a high level by using a dynamic object-oriented programming language and its various advantages, and easy access due to its excellent aptitude for teaching.

[Copyright notice will appear here once 'preprint' option is removed.]

[1] opensource.nokia.com/projects/pythonfors60/

[2] code.google.com/android/

[3] mindstorms.lego.com

[4] mindstorms.lego.com/Overview/NXTreme.aspx

[5] www.hpi.uni-potsdam.de/swa/projects/nxtalk

- We describe the implementation of the NXTalk virtual machine. As opposed to many other dynamic language implementations in constrained environments, NXTalk provides the profound introspective and reflective capabilities typically found in other Smalltalk systems.
- We present an evaluation of the NXTalk VM in terms of execution performance and code size.

The remainder of this paper is structured as follows. In the next section, we will first give a more detailed overview of the NXTalk project and the software and hardware components employed therein. In Sec. 3, the NXTalk VM implementation and programming environment are described, followed by a programming example in Sec. 4. The VM implementation is evaluated in terms of characteristics relevant to embedded systems in Sec. 5. Related work is discussed in Sec. 6, and Sec. 7 summarizes the paper and gives information about future work.

2. Background

The NXTalk project tries to combine the expressiveness of Smalltalk with the Lego Mindstorms NXT robot in order to provide a powerful environment for further research on VMs for embedded systems and end-user programming. In this section, we will give information about the two elements that are combined in NXTalk; namely, the Smalltalk programming language, and the NXT platform.

2.1 Smalltalk and Squeak

Smalltalk is a dynamic object-oriented programming language and programming environment [10]. Its pervasive object metaphor manifests itself in the absence of language-level control structures. Instead, late-bound messages are used throughout the whole system. In order to achieve performance, the compiler optimizes control structures at the bytecode level by using inlining and thus avoiding method invocations for them.

Squeak[6] is an open-source Smalltalk implementation, created to meet the need for an educational development environment [12]. An additional authoring environment using Squeak's and Smalltalk's capabilities is Etoys[7], especially developed for teaching children [1]. While Etoys allows for easy graphical scripting of objects, its underlying Smalltalk base encourages interested learners to further explore the full-featured programming environment. The focus of Squeak—like for most Smalltalk systems—is on writing as much code as possible in Smalltalk itself in order to provide a consistent system for researchers and students. Tools like a debugger featuring hot code-replacement, unit testing and powerful refactoring reduce development time dramatically.

[6] squeak.org

[7] squeakland.org

2.2 Lego Mindstorms NXT

As a low-cost end-user product, the Lego Mindstorms NXT robot comes with very restricted hardware components. Programs running on it are constrained to only 64 kB of RAM, backed up by 256 kB of non-volatile Flash memory. Execution is driven by a 32-bit ARM7TDMI (AT91SAM7S256) running at 48 MHz and an 8-bit Atmel AVR (ATmega48) co-processor at 4 MHz. While the former fulfills the task of program execution and coordination, the latter controls motors and analog sensors.

Up to three motors and up to four sensors can be connected to the NXT. The motors can also serve as rotation sensors. The NXT is delivered with touch, noise, light, and ultrasonic sensors. For further human interaction the NXT brick[8] has four buttons, a 100x64 pixel monochrome LCD display, and a small loudspeaker. A PC-based graphical programming environment called NXT/G is also included. Programs can be transferred to the NXT via USB or Bluetooth.

The NXT has no memory management unit. Instead, Flash memory is hard-wired to be mapped at 1 MB and the RAM at 2 MB. This means that every address pointing into these two regions requires only up to 22 bits, wasting the remaining 10 bits and leaving room for storing payload in references at the cost of some bit masking.

The ARM processor is capable of two execution modes: a normal one and the "Thumb mode" [17]. The former uses four-byte instructions, the latter only two-byte encoded ones. While this reduces code size, it disallows several optimized process-and-shift instructions. Moreover, the ARM requires a 2-byte alignment of data objects.

Flash memory supports writing in blocks of 256 bytes. Reading from Flash is as fast as reading from RAM, but writing to it is ten to a hundred times slower. RAM is thus the preferred place for data modifications and the Flash memory backing storage for immutable data like program code.

2.3 NXOS

NXOS[9] [3] is a small operating system especially written for the NXT system. It provides a thin abstraction layer over the NXT hardware and the necessary device drivers. Furthermore, it includes basic implementations for memory management, scheduling and a Flash file system. One of NXOS' purposes is to be used as a base for higher level software or VMs. Thus, it is highly modularized and components like memory management can be replaced by custom implementations.

3. The NXTalk Environment

NXTalk is divided into two parts. The first is a VM running on the NXT brick, which we tried to make as small as possible in order to leave more room for user applications. Also,

[8] The NXT main unit is, basically being an oversized Lego brick, often affectionately called "the brick".

[9] nxt.natulte.net/nxos/trac

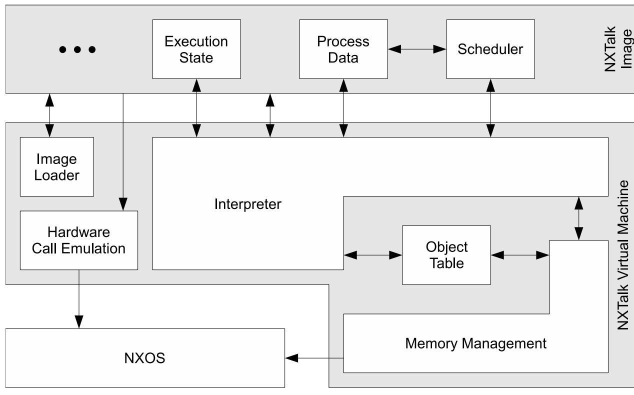

Figure 1. Architecture of the NXTalk Virtual Machine.

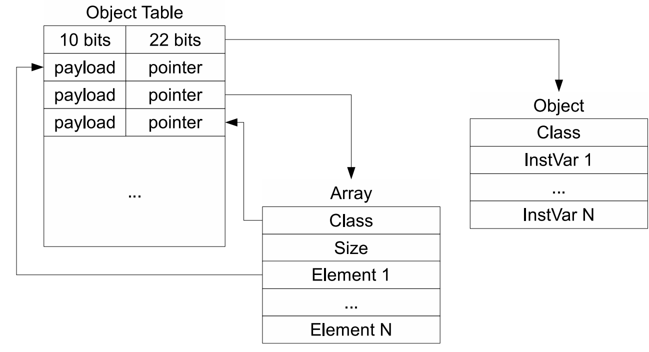

Figure 2. Object layout and object table.

we preferred size optimizations over performance optimizations where they were applicable and did not hinder overall system performance and usability too much. The second is a programming environment built in Squeak that provides class libraries and facilities to build images for transferal to the NXTalk VM on the brick.

3.1 The NXTalk Virtual Machine

NXTalk's VM provides the execution environment for applications deployed to the NXT. Figure 1 shows its overall architecture. The main actor is the execution engine with the interpreter, manipulating the objects in the object space, called image. The object space is managed by the memory management component. It takes care of object allocation and automatic garbage collection. For communicating with the different NXT components like sensors or motors, NXTalk uses NXOS, presented in Sec. 2.3. Since Smalltalk implementations typically store all objects in an image, NXTalk can abandon NXOS' Flash file system: the image is the only "file" that exists, and it is managed directly in-memory. As NXTalk manages object allocation and deallocation on its own, there is no need for using the memory management of NXOS.

Our current implementation of NXTalk including NXOS and its device drivers fits within 33 kB of Flash plus 9 kB of RAM using size-optimizing compiler settings and the ARM Thumb mode.

3.1.1 Object Model

Smalltalk features a very simple object model that allows the VM to treat the deployed applications and libraries as a single collection of objects. Only some special object references have to be stored along with the image, e. g., the image entry point, which is represented in the form of an object and a message. At VM startup, the interpreter is pointed to the startup object as receiver and instructed to send the given message to it in order to start image execution.

In order to achieve our size goals, objects have to be kept very small. Typically, an object consists of a header, a class reference and several instance variables referring to other objects. Due to the ARM 32-bit architecture, every reference would consume four bytes. Nevertheless, it would also waste 10 bits each time (cf. Sec. 2.2). Thus, we decided to store the address of each object only once in an object table [10]. References to objects are then represented as indices into this table. By limiting the maximum number of objects, we can store this index in two bytes. This pays off by a 50 % space saving as soon as an object is referenced twice, because each further reference takes up only two instead of four bytes, thus compensating for the entry in the object table.

Naturally, an object table brings about a certain impact on performance due to the additional indirection it introduces, but the remaining 10 bits can be used to store additional object meta-data, without further dereferencing the real object pointers. For example, object header data like mark bits or reference counts for garbage collection can be moved here, thus eliminating the need for an object header. The exact usage of these bits is described below.

Another data element typically stored in the object header is the size of an object. For instances of classes with a fixed size, this can easily be calculated by reading the class format. Variable-sized instances like arrays still require an extra size field in the object, which is dynamically included for them, as shown in Figure 2.

Tagged Integers The size of a single object containing no instance variables consists of two bytes for the class reference and four bytes for its object table entry, amounting to six bytes. Number objects would require even more memory to store their value. To eliminate this overhead, NXTalk encodes integers small enough directly within the 16-bit object reference. To distinguish between object table indices and integers, we use the least significant bit as a tag, leaving 15 bits for both of them, as visualized in Fig. 3.

If the tag bit is zero, the remaining bits represent the integer in two's-complement format. Otherwise, they are interpreted as an object table index. This zero-tagging technique allows to perform additions and subtractions without prior decoding of the number, because the zeroed tag-bit is neutral to these basic arithmetic operations [11].

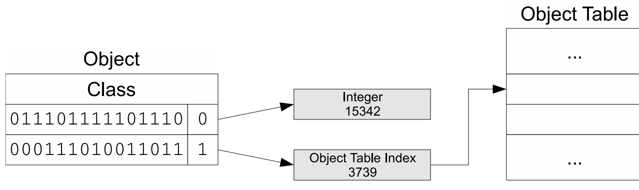

Figure 3. Usage of zero-based tagged integers.

In order to find the address belonging to an object reference, the VM has to shift the 16-bit value right by one to retrieve the object table index. Table lookup itself shifts the index to the left by two because the table entries are four bytes wide. Thus, only a shift by one to the left is needed, if the reference's least significant bit is set to zero before. The `AND` instruction is able to do both operations at once [17] if the ARM is not especially set into Thumb mode.

Using this approach, NXTalk is able to provide integers in the range from -2^{14} to $2^{14} - 1$ without consuming any additional space, while still allowing a hypothetical maximum of 2^{15} objects. This is sufficient given the amount of memory available on the NXT. The NXTalk standard image is 49.6 kB large and contains 3,054 objects. This amounts to an average size of 16.6 bytes per object. Assuming this average, 512 kB of memory would be consumed by objects in case all object table entries were used; i. e., twice as much as the NXT offers. Thus, space exhaustion is not a risk.

Object Layouts In addition to the aforementioned generic object layout, the NXTalk VM needs to know about some special objects and classes. For instance, knowledge of the class layout is necessary, because the class of an object contains information about the size of its instances, its superclass and the messages understood by the object.

Methods and Blocks also are very special objects because they contain elements of variable size. Nevertheless, the format of both is almost identical. Their classes define the element size as one byte, but in fact, they start with a two-byte method header represented by a tagged integer and a list of literal references used in the method or block. Afterwards, the bytecodes of the method follow.

The header itself encodes the counts of arguments and local variables in four bits each, limiting both to a theoretical maximum of 15. Six bits represent the literal count—i. e., the number of literal constants [10] contained in a method—, which corresponds to 63 literals at most. These maximum counts are further restricted by the possible bytecode operands described in Sec. 3.1.3. One last bit indicates that this method is a primitive and uses the ten bits of the temporary and the literal count to encode the number of the primitive.

3.1.2 Memory Management

Automatic memory management for dynamic and object-oriented languages in a constrained environment like the NXT deserves special attention as its performance and behavior directly impact the system's overall usability.

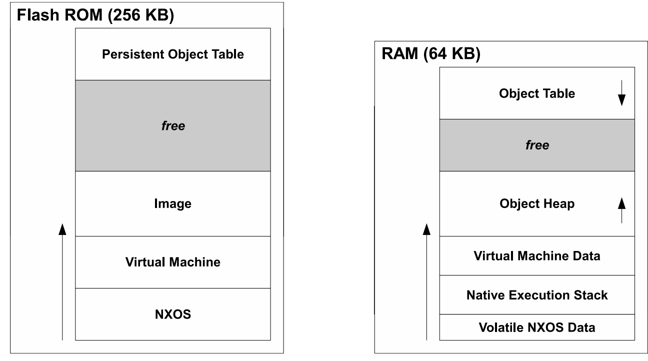

Figure 4. Memory layout on the NXT.

Object Management Basically, NXTalk's object management is responsible for two different entities: the object memory and the object table. While the former grows from lower memory addresses to higher ones, the latter grows from top to the bottom of the available memory and thus towards the object memory. Fig. 4 illustrates this. Furthermore, the object memory is divided into a Flash memory and a RAM part, but the object table is completely stored in RAM. While the Flash memory only contains old and long-living objects like classes and methods including their literals, the RAM part is populated by newly created objects and old objects which have been modified and thus had to be moved to the RAM due to the slow flashing.

Obviously, the Flash part of the object space is modified very rarely, namely at system shutdown—which is when the live object table is copied from RAM to Flash to make it persistent—or application and class deployment. Thus, when we refer to "object space" in the following, it is meant as a synonym for the RAM part.

As already mentioned, new objects are created in RAM. They are allocated at the end of RAM object space as long as there is enough free memory. Vice-versa, the object table grows from the top of the available RAM towards the bottom. NXTalk hence assigns table indices in top-down order beginning at the maximum value.

Although it is quite fast, this technique usually exhausts memory very quickly and both memory regions meet somewhere in the middle. From this point on, a fallback allocation algorithm scans the object space in order to find a suitable object already freed by the garbage collection.

These free objects are managed using free-lists as shown in Fig. 5. If an object is detected not to be referenced any more, its object table entry is marked. Afterwards, the object is inserted into a linked list of free objects. To speed up the lookup process, if free space is needed, NXTalk manages multiple free-lists, which are ordered by the size of their entries. It does so for sizes up to 30 bytes; larger entries are kept in a single list. All in all, there are 16 free-lists, because

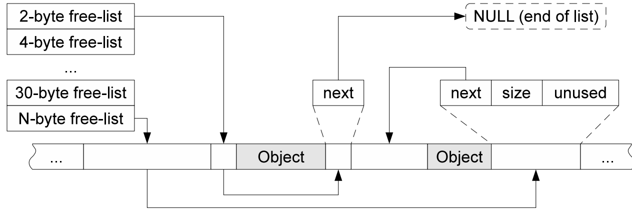

Figure 5. Free-list management.

the minimal size of an object is two bytes, and every object has to be two-byte aligned, as already mentioned in Sec. 2.2.

Inserting a deallocated object into a free-list is as easy as setting the object's first two bytes to the object table index of the previous free-list head and announcing the object as the new head. Thus, object deallocation is done in constant time. However, to avoid fragmentation, the free-list for large objects is always kept sorted by size. Thus, deallocation of objects larger than 30 bytes has linear time complexity depending on the number of already deallocated large objects.

On allocation without free memory at the end of the object space, NXTalk tries to use the free-lists with sizes of at least the required size. If a free entry is found, it is split up if required and the available memory can be used. This allocation scheme was already used by the original Smalltalk-80 implementation [10].

Memory fragmentation is not an issue, as the garbage collector applies heap compaction [13]. If the heap is fragmented and not enough contiguous space is available for an object at allocation time, the allocation request will lead to a collection cycle, resulting in a compacted heap with enough free space.

The described dynamic growth of the object space and the object table allows NXTalk to support different application allocation behaviors. A user program requiring many small modifiable objects has a large object table, but small object space. If it needs only few but big objects, the same RAM region is separated differently. Memory occupied by the object space in the latter case is used by the object table in the former one.

Garbage Collection In order to find orphaned objects, NXTalk uses two well-known algorithms [13]. *Reference counting* provides incremental garbage collection but does not find all orphans. A *mark-and-compact* traversing algorithm eliminates unused objects and avoids fragmentation at the cost of a possibly long system pause.

Upon object assignments, the VM increments or decrements the involved objects' reference counts. The count is stored in five of the free bits of the object table entries as shown in Fig. 2, thus allowing it to grow up to 31. If it reaches this level, reference counting is deactivated for this object. The actual reference count might be much higher or even lower than 31 after that, but the reference-counting collector does not take such objects into account any more.

If the reference count decreases to zero, the object will be automatically deallocated. Also, the reference counts of the objects it references are decremented. Obviously, this may lead to cascaded deallocations which could introduce a full system stop for some time. A possible solution is to simply stop cascading after some amount of time or a specific number of deallocations. We decided to let the programmer control these parameters from within the NXTalk environment.

Reference counting is not able to detect cyclic dependencies of objects. In order to collect such objects—and objects with reference counts larger than 31—and to defragment the object heap, we implemented a mark-and-compact algorithm. Although it halts execution for some time, it is able to definitely collect all unreferenced objects.

Mark-and-compact GC starts with a marking phase, in which all living objects are marked as being reachable in the object table by traversing their object references. The algorithm starts with a root set consisting of the objects currently accessible by the program code—i. e., all elements on the stack including method activation frames.

As NXTalk's GC only collects objects in RAM and to speed up the marking phase, traversal is stopped when encountering Flash memory objects. Therefore, it is necessary to add objects that were copied from Flash to RAM to the root set. Otherwise, the mark phase might miss them, because live objects in RAM might be referenced by objects in Flash only. Forcing the GC to mark these objects as live ensures that no possibly living object is discarded.

Also, traversing the whole object graph can consume large parts of the stack, which might even overflow. Thus, NXTalk uses a slight modification of the Deutsch-Schorr-Waite (DSW) algorithm [16] for non-recursive heap traversal. This algorithm stores the back-reference in the object itself instead of on the stack and iteratively traverses objects. Thus, it does not need any further stack space. In order to remember which path of the list it took down in the traversal, the original DSW algorithm uses one prefix bit. This is enough because it regards list items as being of fixed size with two fields.

NXTalk, however, has variably-sized objects. To keep track of already visited instance fields, NXTalk reuses the reference count bits in the object table entries, as they are not in use during GC and can be restored with their correct value later on. The collector stores the index of the instance variable before continuing traversal for that instance variable. After returning back to the object, it reloads the variable's index from the reference count bits and can find the next instance variable. If the variable's index is too big to fit in the five reference count bits, the value still has to be pushed onto the stack. With this optimization, our marking phase consumes two bytes on the stack for each object having more than 31 references to other objects.

The mark phase is followed by a compaction phase linearly scanning the object heap for marked living objects and copying them to the beginning of RAM. As it also has to update the object table, it needs to find the object table index for a given object address. Searching the object table for each given address would be too expensive, so NXTalk replaces every class reference with the object's table index in a previous link-reversal step. This enables fast access to the object table entry belonging to a specific object address.

In order to be able to restore the class references after compaction, they are saved within object address fields of the object table as these are unused during the compaction phase. Nevertheless, class information for objects is not available during the following linear scan anymore. Thus, the size cannot be retrieved for non-arrayed objects—necessary to compute the address of the following object. NXTalk fills the remaining free bits in the address fields of the object table to store this payload in this step. Additionally, it recalculates the reference counts of living objects during the iteration over the object table.

After link reversal, compaction is trivial: marked objects are copied to the beginning of RAM one by one, overwriting unused objects or free space. Class references and table indices of the objects are updated. Afterwards, control is returned to the interpreter, which continues execution.

This GC implementation's time complexity linearly depends on the number of objects in RAM. However, a general insight is, that 64 kB RAM cannot contain many objects. The theoretical maximum is 32,768 because the minimum size for an object is two bytes for its class reference. This number is reduced by the operating system, some VM code, stack regions, other Smalltalk objects necessary for the execution, and the growing object table.

3.1.3 Execution Model

Smalltalk has a very simple execution model. Nevertheless, due to its many reflective and introspective facilities, several tradeoffs and optimizations have to be made in order to gain enough performance on the NXT. The NXTalk interpreter is stack-based and its main processing units are methods, which are invoked by sending messages to objects. After a correct method to execute is found, a new activation context is set up for it. Afterwards, the interpreter can start to execute the method's bytecodes until it is advised to return to the preceding context or invoke a new one.

Method Lookup If the interpreter encounters a message send, it starts a method lookup. To do so, it fetches the corresponding message name (selector) from the method literals. Afterwards, it determines the position of the receiver object on the stack by examining the argument count of the selector and the receiver's class, which contains the method dictionary, can be fetched. The method dictionary contains association objects, which map from a message selector to a method reference. As these objects are all accessible from Smalltalk code by means of introspection, the interpreter has to dereference them and also have knowledge about the involved object structures on the heap.

If the method is not found in the receiver's class, continuing lookup along the superclass chain may lead to time-consuming method lookups upon every message send. To avoid this, NXTalk uses a method cache like the one used in Squeak. The cache is organized as a hash table; selector and receiver class are used to compute hash keys. Cache entries contain (selector,class,method) tuples. Standard lookup is performed only when the cache misses, and resolved methods are stored in the cache.

Stack Frames In Smalltalk, stack frames, called "contexts", are accessible from everywhere. This allows for code influencing inter-method control flow, e. g., exception handling, to reside on the Smalltalk side entirely. Nevertheless, there is a crucial overhead for the interpreter, because it has to create a new frame object on the object heap upon every method or block invocation.

As this would quickly exhaust memory and cause frequent garbage collections, NXTalk uses a global frame pool. Instead of deallocating unused frames, they are pooled and can be quickly reused later.

Constant-time allocation and deallocation of frames in the pool are achieved by implementing it as a linked list, where the context field acts as link to the next free frame. The pool size does not exceed a certain limit, because a tradeoff between wasting memory and fast frame reification has to be made. This limit should be based upon the typical application stack depth and is configurable.

A sample stack layout is given in Fig. 6. Every participating object is allocated on the Smalltalk heap. The *Current Thread* object links to the frame currently being executed by the interpreter. Frame objects contain meta-information about the method or block invocations they represent. The sender chain of frame objects builds up the execution stack.

In the example, a method `doSomething` sends a message to its receiver, passing a block as argument. The invoked method in turn executes its block argument by sending it the `value:` message. A VM primitive directly allocates a block frame and directs interpreter execution to it.

The *Current Thread* object also links to an array that the interpreter uses as the current stack. Thus, the interpreter uses Smalltalk objects for managing its own execution state while also allowing access to the available metadata. In order to speed up interpretation, the addresses of the involved objects are cached between method invocations, garbage collections and context switches.

Bytecodes The bytecode set of a language implementation affects both execution speed and code size. In NXTalk, due to Smalltalk's consistent syntax, we only need 20 distinct opcodes. We represent opcodes and their operands in a single-byte packed format as shown in Tab. 1, resulting in smaller and easier to interpret code.

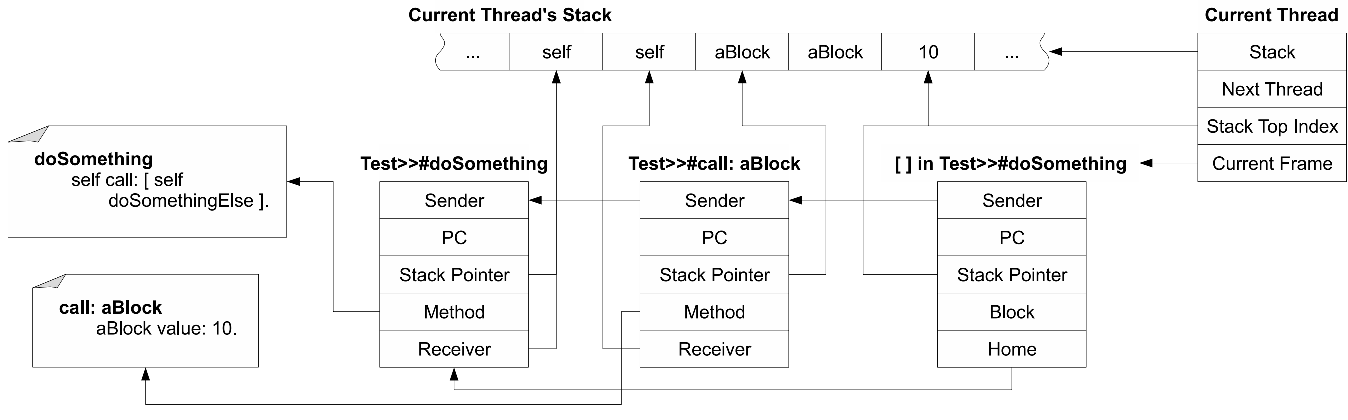

Figure 6. Stack layout in object heap.

Opcode	Representation
Block Return	11111111
Return	00000000
Dup	00000001
Pop	00000010
Push Self	00000011
Push Nil	00000100
Push True	00000101
Push False	00000110
Push thisContext	00000111
Push Argument	00001XXX
Push Temporary	00010XXX
Pop into Temporary	00011XXX
Push Instance Variable	001XXXXX
Pop into Instance Variable	010XXXXX
Push Literal	011XXXXX
Push Block	100XXXXX
Super Send	101XXXXX
Send	110XXXXX
Jump Forward if True	1110XXXX
Jump Backward	1111XXXX

Table 1. The NXTalk bytecode instruction set.

This, of course, limits several aspects of the Smalltalk environment. One of our limitations is the maximum number of local variables and arguments in methods to eight each. Also, we introduced a maximum of 32 literals—i. e., immediate values or symbols used for message sends—per method. This should be adequate for medium-sized methods and automatically encourage the programmer to use small methods, which are considered to be good style in Smalltalk [14].

Small methods also support the two branch instructions, which can jump 15 bytes at most. Backward jumps in fact can only jump one byte less, because the Block Return opcode has the same shape as the maximum backward jump opcode. These two jump instructions are used by the NXTalk compiler for inlining control structures; if jumps are too far, inlining is not used (cf. Sec. 3.2.2).

The final limitation is the possible maximum of 32 instance variables for classes. This is more than sufficient in our view, because a class with too many instance variables indicates a "code smell" [8].

Scheduling While the original Smalltalk-80 implementation [10] used cooperative green threads, NXTalk employs a preemptive model. Although NXOS could provide its own scheduling which provides multiple processes, green threads bear several advantages in our case. First, threads can easily be made a first-class language construct and yet consume less memory. Furthermore, a green thread model is considered to allow for faster context-switching, as less data needs to be restored.

NXTalk's interpreter preempts execution of the current thread on every 1,000 bytecodes and transfers execution to a scheduler thread. This frequency has been determined empirically and proved to be a good compromise, but is configurable from within the environment. The scheduler is also written in Smalltalk and is able to access the active threads as linked list of reified thread objects in the scheduler instance. Every thread contains a reference to its stack and the current stack frame, as shown in Fig. 6.

Being able to exchange the scheduling algorithm via Smalltalk code enables programmers to dynamically adapt system behavior. Also, they do not need to know C in order to add new scheduling behavior to NXTalk. Nevertheless, this openness of the system leads to certain performance impacts, which are documented in Sec. 5.

3.1.4 Interaction with NXOS

While NXTalk does not use NXOS' memory management component, scheduler or file system, we still take advantage of the available and well-documented device drivers for sensor and motor interaction. The "hardware call emulation" building block in Fig. 1 is responsible for communicating with NXOS. Apart from the NXOS features we directly use, the system deals with hardware initialization and system bootstrapping. It sets up interrupts and takes care of

synchronization with the co-processor. This was not in the scope of our work.

NXOS provides simple support for sensor interaction. Pressing a button does not result in an interrupt or event—the NXT hardware architecture does not allow for interrupt semantics. Instead, NXOS and consequently NXTalk have to poll the button state or any other sensor data continuously. For convenient sensor interaction, several VM primitives are provided. They can be called from within the Smalltalk image. Furthermore, threads may register themselves for events from specific buttons or sensors. The virtual machine polls the available data every 500 bytecodes and, if required, interrupts the current thread, continuing execution in a waiting thread. Those blocked threads are not in the process list of the scheduler. Instead, they are in an event-specific linked list, which is known to the interpreter.

3.2 The NXTalk Programming Environment

NXTalk uses Squeak as a programming environment. This allows to reuse Squeak's mature development tools like the class browser or refactoring. A basic image can be built from NXTalk's system libraries and transferred to a NXTalk VM residing on an NXT robot using a simple graphical tool. Afterwards, programmers are able to package their own applications or additional libraries and deploy them on the NXT while NXTalk is running. A compiler transforms the classes created in Squeak to the NXTalk format.

3.2.1 System Libraries

NXTalk provides several dedicated system libraries. Reusing the existing Squeak libraries is not possible because they are too heavy. NXTalk also requires slight modifications to the class hierarchy and several methods. Therefore, any class supposed to be transferred to the robot has to be a subclass of NXTalk's own root class `NXTObject`. Nevertheless, the base classes implement the same protocols as Squeak's libraries do and mirror their hierarchy to some extent.

The NXTalk system libraries provide extensive support for collections, streams and hardware components like sensors, motors or the display. Exception handling is completely implemented in Smalltalk and takes advantage of the stack reflection mechanisms described in Sec. 3.1.3. Concurrency support is based on the hard-wired connection of the scheduler and the VM. While the scheduler is responsible for managing thread objects, the VM uses their data for execution.

An image builder tool invokes NXTalk's packager and compiler and combines the results into a full image runnable by the VM. The image contains the kernel and system packages as well as automatically included special classes required for later package deployment.

3.2.2 Development Support

The NXTalk compiler inspects a Squeak class and converts it and all of its methods into the appropriate NXTalk objects. Being the last component to have access to code before it is transferred to the NXT, the compiler is responsible for code size and performance improvements. As opposed to the NXTalk VM, it has much more resources available in order to perform advanced optimizations. Currently, the compiler does not apply sophisticated analyses, but inlines special message sends assuming they are sent to known receiver classes. This includes loops and conditional clauses. If the jumps needed for these inlining rules are too far to fit into the bytecodes, normal message sends are inserted. This scheme is similar to Squeak's inlining for frequently called methods.

Application deployment is an important issue when creating a robot programming environment. As Squeak does not include an application packaging mechanism, NXTalk provides its own. A package in NXTalk is simply defined by its package description. A description lists all delivered classes, required packages and data objects used by a package. Possible package types are executable applications and add-on-libraries. For convenience, everything installed on the NXT resides in a package. Thus, a base image consists of a kernel, collection, streaming and hardware package.

Once a package is built from its description, compiled classes and data, it is ready for transfer to the NXT. The resulting set of objects is serialized to a data stream using the NXTalk libraries. The stream contents are sent to the NXT, where the running image deserializes the package objects. Once the package classes are registered in the system dictionary, the code is available for further use. The system standard application is able to include all present applications into its menu and present them to end-users.

As already mentioned, NXTalk source code is able to be executed and debugged within Squeak to some extent. In fact, both differ only in created hash values for an object and classes and literal objects like numbers or strings not being already transformed NXTalk objects. Assuming these flaws are non-essential for an application, Squeak can simulate it using graphical front-ends for the display, motors, sensors and other hardware. In order to ease this duality, NXTalk encapsulates hardware access within a special environment object. While this is the system dictionary on the NXT, a simulation engine can be inserted within Squeak.

4. Programming Example

In this section, we will briefly present the hardware configuration and NXTalk source code for a demonstrator robot. The robot is intended to do the following:

- Drive straight ahead until an obstacle is noticed; in that case, stop, back off in reverse direction, and turn away before proceeding.
- Play random tunes at random intervals.
- Shut down if the cancel button is pushed.
- Shut down if the robot bumps into an obstacle that was not noticed.

Figure 7. The Bumper robot.

Name	Role
`radar`	radar sensor object
`touch`	touch sensor object
`tunes`	musical tunes to play
`driver`	driving process
`music`	music playing process
`crashWatcher`	emergency shutdown process
`main`	process waiting for cancel button
`running`	indicate shutdown request

Table 2. Instance members of `NXTBumper`.

Each of these tasks is to be taken out by a separate process.

The robot can be seen in Fig. 7. It features an ultrasonic sensor for obstacle detection (connected to port 3), a touch sensor for triggering emergency shutdown (port 1), and two motors (ports B and C).

The application is called "Bumper", and it is implemented in the class `NXTBumper` inheriting from `NXTDriver` and having the instance members enumerated in Tab. 2. The `NXTDriver` super class provides certain methods for robots with motors connected to ports B and C, allowing them to drive forward and backward as well as to turn around.

4.1 Running

The code for the methods starting up and shutting down the Bumper application is shown in Lst. 1. The `setup` method is most interesting; it initializes sensor objects at the appropriate ports (see above) and also configures the array of musical tunes that can be played. For optimization purposes, the tunes are stored as associations with the desired speed as their key and the actual tune as their value. The tunes are represented in strings whose format roughly corresponds to

```
1  run
2    self
3      setup;
4      spawnProcesses;
5      shutdown
6
7  setup
8    radar := (NXTRadar port: 3) enable.
9    touch := (NXTTouchSensor port: 1) enable.
10   tunes := (NXTArray new: 2)
11     at: 1 put: 180 -> '>8cdedcdepcp2cp';
12     at: 2 put: 120 -> '>2c4dd3c<8a2f';
13     yourself
14
15 shutdown
16   radar disable.
17   radar := nil.
18   touch disable.
19   touch := nil
```

Listing 1. Methods for Bumper startup and shutdown.

```
1  spawnProcesses
2    running := true.
3    driver := [ self drive ] fork.
4    music := [ self music ] fork.
5    main := [ self waitForEnd ] fork.
6    crashWatcher := [ self crashWatch ] fork.
7    [ running ] whileTrue: [ 500 milliSeconds wait ]
```

Listing 2. Spawning the Bumper processes and main loop.

```
1  drive
2    [ true ] whileTrue: [
3      self driveAhead.
4      radar waitUntilBelow: 20.
5      self
6        backOff;
7        turnAway ]
8
9  backOff
10   self driveBackwards.
11   (1000 + 500 atRandom) milliSeconds wait
12
13 turnAway
14   2 atRandom = 1
15     ifTrue: [ self left ]
16     ifFalse: [ self right ].
17   (500 + 500 atRandom) milliSeconds wait
```

Listing 3. Code for driving.

that of `PLAY` commands in implementations of the BASIC programming language.

The code for `spawnProcesses` is shown in Lst. 2. This method simply starts up the four desired processes and then goes to sleep, waking up every 500 ms to check whether a shutdown has been requested.

4.2 Driving

Lst. 3 contains the code relevant to the driving process. The `drive` method essentially consists of an infinite loop that instructs the motors to drive ahead (the method `driveAhead` is inherited from `NXTDriver`) and then goes to sleep until the ultrasonic sensor reports an obstacle at a distance of less than 20 cm.

```
music
  | spk tune |
  spk := NXTSpeaker new.
  [ true ] whileTrue: [
    (500 * 10 atRandom) milliSeconds wait.
    tune := self tunes: 2 atRandom.
    spk
      speed: tune key;
      play: tune value ]
```

Listing 4. Music playing process.

```
waitForEnd
  NXTButton waitForButton: NXTButton exit.
  Processor withoutScheduling: [
    crashWatcher terminate.
    driver terminate.
    self stop.
    music terminate.
    running := false ]

crashWatch
  touch waitUntilPressed.
  Processor withoutScheduling: [
    driver terminate.
    self stop.
    main terminate.
    music terminate.
    running := false ]
```

Listing 5. Shutdown watcher processes.

In case the sensor triggers, the process is woken up and instructs the motors to back off and turn away—both actions are taken out with random values to avoid overly static behavior of the robot.

4.3 Playing Music

The code for the music player process is shown in Lst. 4. The process has a local variable representing the NXT's piezo speaker. It also runs in an infinite loop, going to sleep for a random interval and playing a random tune when waking up.

4.4 Shutdown Watchers

The two remaining processes are those waiting for either the cancel button or the touch sensor triggering. Their code is shown in Lst. 5. They exhibit a largely similar structure. Both immediately go to sleep, waiting for their respective events. When woken up, they disable scheduling, shut down all other processes, and set the `running` control variable to `false` to indicate that the application may now terminate.

4.5 Application Description

Finally, a class is needed that describes the application in terms of classes to deploy, name to display in the NXTalk application menu on the NXT, and entry point; i. e., which message to send to an instance of the application class to start it up. The `NXTBumperApp` class, inheriting from `NXTApplication`, achieves this by providing the appropriate message implementations as shown in Lst. 6.

```
classes
    ↑ #( NXTBumper NXTDriver )

name
    ↑ 'Bumper'

entrypoint
    ↑ #NXTBumper -> #run
```

Listing 6. Deployment support methods.

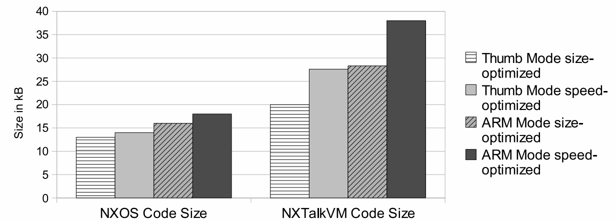

Figure 8. Native code sizes (in kB) of NXOS and the NXTalk VM with different compiler settings.

The application can, for instance, be deployed on an NXT brick connected via USB by evaluating the following statement in a workspace.

```
NXTBumperApp deployOn: NXTUsbConnection connect
```

5. Evaluation

Our general intention in developing NXTalk was to favor size over speed to accomodate the NXT's strict memory constraints. In this section, we will evaluate and discuss several aspects of the implementation.

The memory footprint of NXTalk and applications running on top of it deserve special attention. We evaluate the native and byte code sizes of NXTalk's components and compare their variation with different compiler settings.

Code Size Four components are relevant with regard to the code size of the NXTalk system. NXOS as the underlying operating system is the first; the NXTalk VM executable, the second. Both of them are compiled to native ARM code. The NXTalk system libraries and deployed applications, represented in bytecodes, are the two other components.

Fig. 8 shows the native code sizes of NXOS and the NXTalk VM employing different compiler settings. NXOS compiles down to 12 kB because not every part of it is used—e. g., NXTalk implements its own scheduling and does not require a Flash filesystem. Still, disabling Thumb mode does not significantly increase NXOS' code size, although the instruction size is doubled. Even perfomance optimization result in code only 40 % larger than the smallest version.

This is different for the NXTalk VM. Especially the optimized ARM mode version increases code size significantly. We suppose this is due to the heavy use of inlineable functions instead of macros in our C source code. While macros

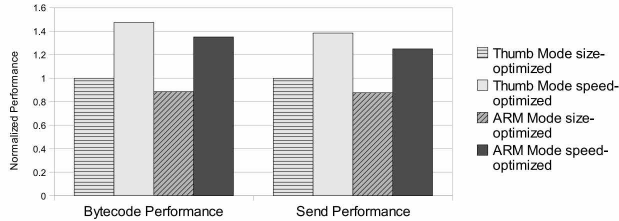

Figure 9. Comparison of different size-performance-tradeoffs with ARM and Thumb mode.

are always expanded even if the compiler is instructed to optimize for size, inlineable functions are only inlined when optimizing for speed.

The NXTalk system libraries consume 33 kB of Flash memory, leaving room for 192 kB of application code minus the space needed for the persistent object table. For instance, a simple, but typical standard application like driving around and turning to other directions on encountering obstacles consumes about 2 kB of Flash memory and around 50 objects (method activations) in RAM.

RAM Usage All of the differently optimized versions presented above consume some amount of RAM for native execution stacks and VM data. Each of them consumes 7–9 kB, thus leaving 55–57 kB for user applications.

As described in Sec. 3.1.3, method and block activations are objects. To use the frame pool more efficiently, they all have the same size, even though a method frame has more instance variables. The frame pool sets the correct class references. A method frame holds eight instance variables. Together with the class reference, a frame consumes 18 bytes. Furthermore, every frame object needs four additional bytes in the object table, resulting in a total of 22 bytes.

Each thread in NXTalk consumes five references, including its class, resulting in 10 bytes, and four bytes in the object table. Additionally, a stack array object is created, which initially consists of 50 2-byte slots and can grow dynamically as needed. This adds 108 more bytes. Thus, a newly created thread including its stack consumes 144 bytes.

Our experiments have shown that a typical application uses 2,000–3,000 long-living objects in Flash and 200–500 in RAM. This includes the base libraries.

Performance Fig. 9 shows a performance comparison of the NXTalk ARM and Thumb mode versions using size and performance optimizing compiler settings. We used two micro-benchmarks from Squeak. The first is a bytecode-heavy benchmark and thus tests raw interpretation performance; the second stresses the interpreter by performing almost only message sends, resulting in many method lookups. The values in the diagram are normalized to the perfomance of the size-optimized Thumb mode version.

As visualized, ARM mode drops below Thumb mode in both cases. This was to our suprise as we expected ARM mode to outperform Thumb mode, which needs more instructions for the same operations. Thus, we decided to continue to use the Thumb mode for NXTalk as it saves code size and provides better performance.

We also measured the time needed for a context switch starting from the invocation of the Smalltalk scheduler thread to the activation of the new thread. On average, this takes 3 ms. By default, context switches appear every 1,000th bytecode (around every 270 ms).

Another interesting measurement is the latency of event handling (cf. Sec. 3.1.4). To measure this, several randomly activated button presses were simulated in the VM. The average latency between the simulated press and the VM noticing it through polling was below 1 ms. As the NXT's system timer has a resolution of 1 ms, we could not measure this latency more accurately. In summary, after the VM notices an event, a waiting thread is awakened after a normal thread scheduling, i. e., after about 3–4 ms.

GC Behavior Regarding garbage collection, duration and frequency of full system pauses must be minimized. Although reference counting distributes object deallocation over time, full GC cycles are needed to compact RAM and collect cyclic structures. Their cost is relevant, so we measured it using an application allocating many small objects in cyclic structures and thus enforcing a compact phase. In this application, memory exhaustion occurs about every 4 seconds, and a full GC cycle takes around 60 ms.

Portability The implementation of the NXTalk VM targets the NXT platform, but its design is not restricted to that. While NXTalk contains several explicit references to NXT hardware drivers, the latter are provided by NXOS, which could be replaced by another hardware abstraction layer.

Memory management in NXTalk makes some assumptions about memory layout on the NXT platform, i. e., it contains some hardwired assumptions about the separation of the available address space in RAM and Flash. Then again, such a separated memory structure is common on embedded platforms, so adapting the memory management logic to support another layout is feasible.

The interpreter is portable. It makes no assumptions about the hardware architecture.

In a nutshell, porting NXTalk to another embedded platform should—as long as the target platform is an ARM architecture—not be thwarted by the VM design as such.

6. Related Work

We discuss related work in two strands: programming environments for the NXT, and other dynamic language implementations for resource-constrained embedded systems.

6.1 NXT Programming Environments

NXT/G is the PC-based standard programming environment delivered with the NXT. Based on LabView, it provides a beginner-friendly graphical programming language. Low-level control and text-based programming are not offered.

LeJOS[10] is a Java environment providing a tiny JVM accompanied by a comprehensive class library as well as several tools to load and run Java programs on the NXT. In order to fit into the available space, it completely abandons reflective capabilities. In contrast, NXTalk provides Smalltalk's full reflection features allowing for more introspection and intercession than possible in any Java.

pbLua[11] makes the Lua scripting language[12] available on the NXT. A dedicated firmware starts a Lua shell on the NXT, which receives commands from a terminal emulator on the PC. It is also capable of compiling, storing and running full Lua scripts on the NXT. Lua is a dynamically typed language combining multiple paradigms like imperative and functional programming. It provides a small set of built-in features allowing for easy implementation of, for instance, object-orientation. While it is possible to use an IDE such as Eclipse[13] in order to edit Lua source code and integrate a terminal connected to the NXT, a simulation or testing environment on a PC is not available. Unlike NXTalk, code addressing NXT functionality has to be executed on a connected NXT or at least stored there for later execution.

RobotC[14] and NXC[15] are C-like languages. Their IDEs are powerful and provide tools like interactive and remote debugging, but simulation without an NXT is not possible.

None of these environments bridge the gap between being beginner-friendly and allowing more professional users to program in a high-level language matching the predominant object-oriented paradigm. NXTalk achieves this by making Smalltalk, an easy-to-learn and yet powerful and expressive object-oriented language, available on the NXT.

6.2 Dynamic Languages

Concerning dynamic language VMs for resource-constrained embedded systems, our focus is on Smalltalk. Moreover, we also revisit further Java VMs targeting similar systems.

The Resilient Smalltalk[16] platform [15, 2] bears many similarities with NXTalk: a Smalltalk VM running on an embedded device is fed with applications and data from an IDE running on a PC. In the case of Resilient, Eclipse is used as IDE. The Resilient VM consumes 128 kB of RAM. Differences between Resilient and NXTalk can be found in these regards. NXTalk's memory consumption is much more constrained: it has only half as much RAM available, but can rely on an additional 256 kB of Flash memory. While this is comfortable, it requires a sophisticated memory management scheme that organizes objects according to their changing over time. Also, the NXTalk programming environment is written in a superset of the language that NXTalk is able to execute. This allows for straightforwardly testing NXTalk programs in the IDE: it is not required to start a dedicated NXTalk instance.

Pocket Smalltalk[17] is an IDE and VM targeting PalmOS. Its executable, containing the VM and standard class libraries, fits within 25 kB, while NXTalk needs 52 kB. These small code sizes were possible, because PalmOS was a 16-bit platform. Although NXTalk uses the ARM Thumb Mode and its packed two-byte instruction format, it still needs 32-bit data pointers, which significantly increases code size. However, Pocket Smalltalk did not support any kind of remote debugging or simulation and was abandoned in 2002.

ENVY/Embedded is a commercial Smalltalk system created by OTI [6] providing several features similar to NXTalk. It included interactive remote-debugging, image packaging and images running partly in ROM. ENVY/Embedded could target various embedded systems down to 512 kB of memory.

SqueakNOS[18] is a Smalltalk implementation running directly on hardware, without an operating system in between. Few elements of the system are implemented in C and assembler. SqueakNOS is not targeted at embedded systems, but at general-purpose PCs. The principle of implementing the entire system in (mostly) one language is intriguing, but in the course of developing NXTalk, we have preferred the chosen solution because it allowed us to address primitives for, e. g., sensor and motor interfacing that are already offered by NXOS, instead of implementing them ourselves.

Apart from LeJOS, several other Java VMs for embedded systems exist. Sun's CLDC HotSpot[19] fits into 300 kB of memory; its predecessor KVM[20], based on the Spotless research JVM [19], only needed 128 kB. They achieve these sizes by abandoning support for some reflective capabilities and limiting class libraries. An underlying operating system is required, though, consuming additional space. NXTalk, conversely, needs 64 kB *including* NXOS. However, the CLDC HotSpot features JIT compilation and exploits ARM hardware Java optimizations.

The Squawk JVM targets devices similar to the NXT in terms of memory constraints [18]. It is written in Java and fits on SunSPOT devices[21] equipped with an ARM9 processor, 512 kB RAM and 4 MB Flash. As it is also available for desktop PCs, it is possible to simulate programs before deploying them to the embedded system.

In 2008, Google released its Android mobile phone platform based on their Dalvik VM[22]. They compile Java byte-

[10] `lejos.sourceforge.net`

[11] `hempeldesigngroup.com/lego/pbLua`

[12] `www.lua.org`

[13] `www.eclipse.org`

[14] `www.robotc.net`

[15] `bricxcc.sourceforge.net`

[16] Resilient Smalltalk is also known as OOVM.

[17] `www.pocketsmalltalk.com`

[18] `sourceforge.net/projects/squeaknos`

[19] `java.sun.com/j2me/docs/pdf/CLDC-HI_whitepaper-February_2005.pdf`

[20] `java.sun.com/products/cldc/wp/KVMwp.pdf`

[21] `www.sunspotworld.com`

[22] `sites.google.com/site/io/dalvik-vm-internals`

codes into a Dalvik-special, size-optimized bytecode format. The VM saves memory by sharing code like the standard class libraries between multiple running instances. Nevertheless, Android targets embedded devices very different from NXTalk's constraints: 64 MB total system RAM and Linux as underlying operating system allow for much larger applications than possible with NXTalk.

URBI[23] [4, 5] aims at providing a common abstraction for robot programming, featuring a dedicated language and run-time environment. The latter is comparatively heavyweight, encompassing a component model and client-server infrastructure. URBI is available for many diverse robotics platforms, including the NXT.

7. Summary and Future Work

NXTalk is a Smalltalk programming environment for the Lego Mindstorms NXT consisting of a very small VM and an integrated development environment based on Squeak. It provides the full range of Smalltalk programming abstractions, including reflective capabilities, while imposing only few restrictions on programmers. In a larger context focusing on providing end-user programming capabilities in the embedded systems domain, NXTalk forms the technical basis and thus a first core contribution.

Future work will cover improvements of end-user experience and further VM research. To achieve the former, high-level, beginner-friendly aspect, a full Etoys [1] implementation utilizing NXTalk is planned in order to provide a graphical programming environment based upon Squeak. This will allow programming beginners to implement robot applications via Etoys and explore object-oriented programming possibilities. Also, simulation support within Squeak for NXTalk will be improved. In conjunction with an Etoys front-end, a full two-dimensional graphical simulation will be possible.

Another important enhancement is the availability of remote debugging and profiling code running on the robot. Implementing this is straightforward as it can take advantage of Smalltalk's meta-programming facilities. Programmers reduce turn-around times and thus raise their productivity by avoiding numerous deploy/trial/fix cycles.

A known problem of dynamic memory-managed languages is their hardly predictable real-time behavior. Future work will incorporate considerations of NXTalk's real-time applicability. This also includes different garbage collection algorithms with better predictability.

A just-in-time compiler could definitely speed up NXTalk. While typical JIT compilers require large amounts of additional memory to store the methods in native code, hybrid JIT compilers like HotpathVM compile only frequently used execution traces instead of methods and have a very small memory footprint [9].

The NXTalk VM itself is also a target for optimizations. We want to investigate possibilities to use the Flash memory more efficiently. For instance, rarely modified objects could migrate to Flash leaving more space in RAM for new objects. Also, the object table is currently completely stored in RAM, although its entries for Flash objects are updated only in some rare cases. By separating the object table into a RAM and a Flash part and storing the latter in the Flash memory, further memory gains are possible.

Acknowledgments

The authors are grateful to Michael Engel for his comments and valuable suggestions during the writing of this paper.

References

[1] B. J. Allen-Conn and K. Rose. *Powerful Ideas in the Classroom*. Viewpoints Research Institute, Inc., 2003.

[2] J. R. Andersen, L. Bak, S. Grarup, K. V. Lund, T. Eskildsen, K. M. Hansen, and M. Torgersen. Design, implementation, and evaluation of the Resilient Smalltalk embedded platform. *Computer Languages, Systems & Structures*, 31:127–141, 2005.

[3] D. Anderson and J. Flesch. Développement d'un système d'exploitation embarqué pour le LEGO Mindstorms NXT. Technical report, Université de Technologie Belfort-Montbéliard, 2007.

[4] J.-C. Baillie. Universal programming interfaces for robotic devices. In *sOc-EUSAI '05: Proceedings of the 2005 joint conference on Smart objects and ambient intelligence*, pages 75–80. ACM, 2005.

[5] J.-C. Baillie. Urbi: Towards a universal robotic low-level programming language. In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3219–3224. IEEE, 2005.

[6] K. Clohessy, B. M. Barry, and P. Tanner. New Complexities in the Embedded World - The OTI Approach. In *ECOOP '97: Proceedings of the Workshops on Object-Oriented Technology*, pages 472–481, London, UK, 1998. Springer-Verlag.

[7] A. Dotts and D. Birkley. Development of reusable test equipment software using Smalltalk and C. *SIGPLAN OOPS Mess.*, 4(2):31–35, 1993.

[8] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[9] A. Gal, C. W. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proc. VEE '06*, pages 144–153, New York, NY, USA, 2006. ACM.

[10] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[11] D. Gudeman. Representing Type Information in Dynamically Typed Languages. *THE UNIVERSITY OF ARIZONA, vol. TR*, pages 93–27, 1993.

[12] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: the story of Squeak, a practical Smalltalk

[23] `www.gostai.com`

written in itself. *SIGPLAN Not.*, 32(10):318–326, 1997.

[13] R. Jones and R. Lins. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[14] E. Klimas, D. Thomas, and S. Skublics. *Smalltalk with style*. Prentice Hall, Englewood Cliffs, NJ, 1996.

[15] K. V. Lund and J. R. Andersen. Revolutionizing Embedded Software. Master's thesis, Department of Computer Science, University of Aarhus, Denmark, 2003.

[16] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM*, 10(8):501–506, 1967.

[17] D. Seal. *Arm Architecture Reference Manual*. Addison-Wesley Professional, 2001.

[18] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the bare metal of wireless sensor devices: the squawk Java virtual machine. In *Proc. VEE '06*, pages 78–88, New York, NY, USA, 2006. ACM.

[19] A. Taivalsaari, B. Bush, and D. Simon. The Spotless System: Implementing a Java™System for the Palm Connected Organizer. *Sun Microsystems Laboratories Technical Report SMLI TR-99-73, February*, 1999.

[20] D. Thomas. Ubiquitous applications: embedded systems to mainframe. *Commun. ACM*, 38(10):112–114, 1995.