PICOBIT: A Compact Scheme System for Microcontrollers

Vincent St-Amour
Université de Montréal
stamourv@iro.umontreal.ca

Marc Feeley
Université de Montréal
feeley@iro.umontreal.ca

Abstract

Due to their tight memory constraints, small microcontroller based embedded systems have traditionally been implemented using low-level languages. This paper shows that the Scheme programming language can also be used for such applications, with less than 7 kB of total memory. We present PICOBIT, a very compact implementation of Scheme suitable for memory constrained embedded systems. To achieve a compact system we have tackled the space issue in three ways: the design of a Scheme compiler generating compact bytecode, a small virtual machine, and an optimizing C compiler suited to the compilation of the virtual machine.

1. Introduction

Applications for embedded systems vary greatly in their computational needs. Whereas some modern cell phones, GPS receivers, and video game consoles contain CPUs, memory and peripherals that are comparable to desktop computers, there is at the other extreme embedded systems with very limited resources. We are interested in applications with complex behavior and low speed requirements such as smart cards, remote sensors, RFID, and intelligent toys and appliances. These devices have relatively simple, slow, power efficient processors and only a few kilobytes of memory integrated with peripherals on an inexpensive single chip microcontroller.

Due to the extreme memory constraints such applications are traditionally implemented using low-level languages, typically C and assembler, which give programmers total control and responsibility over memory management at the expense of software development ease and speed. The overall objective of our work is to show that a high-level mostly functional garbage collected language is a viable option in this context. In this paper we explain the design of the PICOBIT system, a very compact implementation of the Scheme programming language which targets these applications. We discuss three variants of the system, which represent different trade-offs and levels of featurefullness. The most compact variant allows Scheme programs to run on microcontrollers with less than 6 kB of ROM and 1 kB of RAM. The system is being used in two notable contexts. It is the firmware of the “picoboard”, a small mobile robot programmable in Scheme which is used to teach introductory computer science at the Université de Montréal. It is also used to implement the S^3 network protocol stack [10], which implements a basic stack for embedded systems supporting TCP, UDP, ARP, etc.

2. Related Work

Virtual machine-based approaches have been used in the past to run high-level languages in embedded environments. Invariably space savings are achieved by implementing a subset of an existing high-level language. For example, the Java language has been adapted for embedded applications and the most compact version is the Java Card Platform virtual machine [11]. To reduce the memory requirements some important features of Java have been removed, notably garbage collection and the 32 bit integer type (`int`) are optional, and the 64 bit integer type (`long`) and threads do not exist. Therefore the programming style is lower-level than with full Java. Moreover smart cards which run Java typically have an order of magnitude more memory than our target platforms.

Due to its small size Scheme has been a popular language to implement in memory constrained settings. Many of the compact systems are based on interpreters and were designed for workstation class platforms. Some of the most compact are based on a compiler generating compact bytecode for a virtual machine. In particular the BIT [3] and PICBIT [6] Scheme systems implement most of the R4RS [2] and target small embedded systems having less than 8 kB of RAM and less than 64 kB of ROM.

3. Overview

The PICOBIT Scheme system has three parts: the PICOBIT Scheme compiler, the PICOBIT virtual machine, and the SIXPIC C compiler. The PICOBIT Scheme compiler runs on the host development system, which is typically a workstation, and compiles from Scheme to a custom bytecode designed for compactness. The Scheme compiler is itself written in Scheme, though it is not self-hosting.

The PICOBIT VM runs on any platform for which there is a C compiler. Currently, we target the popular Microchip PIC18 family of microcontrollers which are cheap single-chip microcontrollers. The VM executes the bytecode produced by the PICOBIT Scheme compiler. The VM is written in C for portability reasons, since most microcontroller platforms already have C compilers targeting them. Therefore, the PICOBIT virtual machine can be compiled for any microcontroller which has a C compiler, making PICOBIT a highly portable platform.

Finally, we have developed the SIXPIC C compiler, a C compiler which was designed specifically to compile virtual machines. We studied the patterns present in typical virtual machines (and the PICOBIT virtual machine in particular) to add specialized optimizations and omit certain features of the C language in order to reduce the size of the generated code for virtual machines. This compiler is typically used to compile the PICOBIT virtual machine.

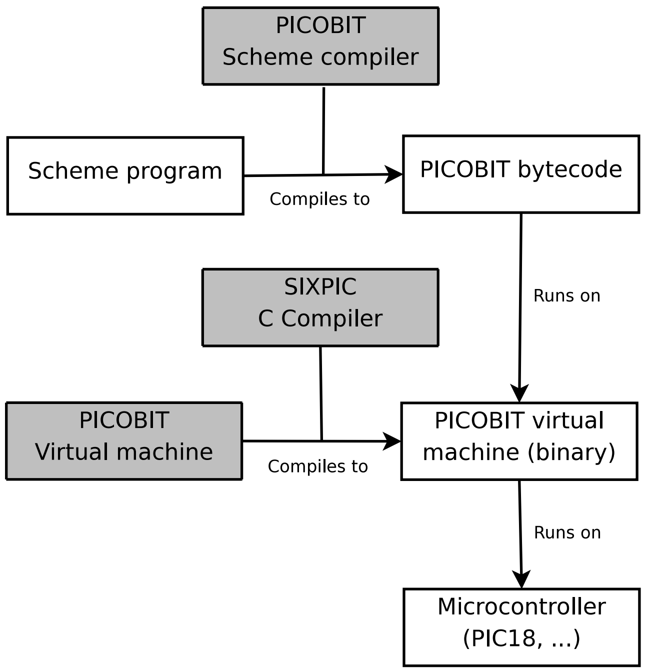

Figure 1. Workflow of the PICOBIT Scheme system

4. General Approach

Because of the code size limitations of our target environment, our approach was designed with the primary goal of generating compact code.

The bytecode the PICOBIT Scheme compiler generates is higher level than raw machine code. The bytecode necessary to accomplish a task is typically more compact than the corresponding machine code. Therefore, the use of interpreted bytecode can lead to savings in a program's code size over the use of machine code. We must keep in mind that the virtual machine needed to execute this bytecode also takes space. However, since the size of the virtual machine is independent of the size of the programs it executes, it is a fixed cost that is amortized over the cost of all the executed programs. We therefore postulate that once applications reach a certain size, the combined sizes of the application's bytecode and of the virtual machine would be smaller than the size of the machine code resulting from the native compilation of the application.

Another key point of our approach is that we control every step of the execution process. By controlling both the Scheme compiler and the virtual machine, we can adapt the bytecode representation to better fit the needs of our applications. For instance, some operations that occur frequently in the applications compiled by the PICOBIT Scheme compiler can be implemented directly as instructions in the virtual machine for efficiency and code size reasons. This was done in the PICOBIT virtual machine for some high-level vector operations that were often used by applications, including the S^3 [10] network stack.

Controlling both the virtual machine and the C compiler which compiles it means that we can specialize the C compiler to use domain-specific optimizations: optimizations that are especially interesting when compiling virtual machines or optimizations that are possible thanks to properties of virtual machines, and would not be valid for all programs.

Finally, the use of a virtual machine also increases the portability of our system. Since the PICOBIT virtual machine is written in a highly portable subset of C, porting it to different architectures is easy. So far, PICOBIT has been ported to the PIC18, MSP430, i386, amd64 and PowerPC architectures, and compiles successfully using the SIXPIC, MCC18, Hi-Tech C, mspgcc, and gcc C compilers. Of course, this portability argument does not yet extend to our SIXPIC C compiler, which currently only supports the PIC18 architecture.

Several versions of the PICOBIT Scheme system exist, catering to different application types and sizes. The full version of PICOBIT supports all the features described in this article, and is suitable for large applications dealing with a large amount of data. A somewhat smaller version of PICOBIT removes support for unbound precision integers in return for a smaller virtual machine size. Finally, a minimalist version of PICOBIT also exists, called PICOBIT Light, which removes support for unbound precision integers and byte vectors, is limited to 16 global variables and 128 memory objects, but is much more compact than the full version (5.2 kB versus 15.6 kB). This version is appropriate when building simpler applications that only deal with small amounts of data at the same time. For example, a temperature sensor that sends reports via UDP using the S^3 network stack.

5. Supported R5RS Scheme Subset

Unlike most programming platforms targeting embedded systems, PICOBIT supports a large number of high-level programming language features. It supports a broad subset of the R5RS [7] Scheme standard including :

- Macros
- Automatic memory management
- Lists
- Byte vectors
- Closures and higher-order procedures
- First-class continuations
- Lightweight threads
- Unbound precision integers

Other features were consciously excluded due to their lack of usefulness in an embedded context :

- Floating-point, rational and complex numbers
- String to symbol conversion (and vice versa)
- S-expression input
- File I/O
- `eval`
- Vectors are implemented with lists (they are not a distinct type)

5.1 Unbound Precision Integers

PICOBIT supports arithmetic on unbound precision integers, whereas most embedded C compilers limit integers to 16 or 32 bits. The support for large integers in embedded systems can create opportunities to do processing that would traditionally be done on host systems or specialized hardware (such as cryptographic calculations or counting elapsed time to a high resolution) directly on microcontrollers, therefore reducing latency and bandwidth needs, and increasing the autonomy of such embedded systems.

The implementation of unbound precision integers has been done inside the PICOBIT virtual machine. At the bytecode level, arithmetic operations on large integers are indistinguishable from operations on small integers, as both use the same set of instructions, the dispatch between small and large integers being done by the virtual machine. This reduces the number of instructions the virtual machine needs to support, which both reduces the size of the virtual machine, and increases the compactness of the bytecode.

The implementation of unbound precision integers in the VM is detailed in section 7.5.

5.2 Built-in Data Structures

Being a member of the LISP family of languages, the Scheme language makes heavy use of lists. Therefore, PICOBIT offers built-in support for lists and implements many common list operations. It is worth noting that these lists are heterogeneous lists, and can thus be used to implement most other data structures easily.

This flexibility opens possibilities regarding which classes of applications can reasonably be implemented in embedded systems. Indeed, some applications which have been deemed too complex for small embedded systems would be straightforward to implement using advanced data structures, reducing the need for more sophisticated hardware where microcontrollers could suffice.

In addition to lists, PICOBIT offers support for byte vectors, which are equivalent to fixed-width byte arrays. Vectors being more efficient than lists for many tasks common on embedded systems (mostly thanks to their $O(1)$ random access), vector support is especially interesting on our target platforms. The implementation of byte vectors in the VM is explored in detail in section 7.8.

Finally, in addition to the aforementioned data structures, PICOBIT also offers limited support for strings. While advanced string-processing operations, such as regular expressions, are not built-in (and would be costly in terms of code size to implement), the operations supported by PICOBIT are useful for debugging and for working with simple text-based communication protocols. For example, the S^3 TCP/IP stack, built on PICOBIT, includes a simple web server that uses PICOBIT's string primitives extensively. Of course, since embedded systems are unlikely to be used for advanced text-processing, this lack of advanced string operations is not, in our opinion, much of a problem.

5.3 First-Class Continuations

First-class continuations are one of Scheme's key features, and accounts for a large part of the language's flexibility. They are usually considered difficult, or costly, to implement, which has led some Scheme implementations to omit them.

Since first-class continuations can be used to implement useful control structures that cannot easily be implemented using traditional embedded development techniques (such as multithreading), we chose to implement them in PICOBIT. To illustrate this, the PICOBIT standard library includes a compact continuation-based multithreading system, shown in full in figure 2. Writing such a multithreading system in C and including it in the virtual machine would have likely resulted in a larger code size. In addition, the same first-class continuation primitives used here could be used to implement backtracking or early exits without any changes to the virtual machine.

PICOBIT provides a first-class continuation API similar to the one proposed in [5], which consists of three procedures. The call `(get-cont)` returns the continuation object of the current procedure, which can then be used with the other two procedures. The call `(graft-to-cont` *cont thunk* `)` calls *thunk* with continuation *cont*. The call `(return-to-cont` *cont val* `)` returns *val* to the *cont* continuation.

The `call/cc` procedure is also provided in the standard library. As shown in figure 2 it is implemented using the above primitives.

```
(define root-k #f) ;; root (empty) continuation
(define readyq #f) ;; queue of runnable threads

(define start-first-process
  (lambda (thunk)
    (set! root-k (get-cont))
    (set! readyq (cons #f #f))
    (set-cdr! readyq readyq)
    (thunk)))

(define spawn
  (lambda (thunk)
    (let* ((k (get-cont))
           (next (cons k (cdr readyq))))
      (set-cdr! readyq next)
      (graft-to-cont root-k thunk))))

(define exit
  (lambda ()
    (let ((next (cdr readyq)))
      (if (eq? next readyq)
          (halt)
          (begin
            (set-cdr! readyq (cdr next))
            (return-to-cont (car next) #f))))))

(define yield
  (lambda ()
    (let ((k (get-cont)))
      (set-car! readyq k)
      (set! readyq (cdr readyq))
      (let ((next-k (car readyq)))
        (set-car! readyq #f)
        (return-to-cont next-k #f)))))

(define call/cc
  (lambda (receiver)
    (let ((k (get-cont)))
      (receiver
       (lambda (r)
         (return-to-cont k r))))))
```

Figure 2. Multithreading system built using PICOBIT's first-class continuations, and implementation of `call/cc`

6. The PICOBIT Scheme Compiler

The PICOBIT Scheme compiler is a specialized optimizing Scheme compiler which generates bytecode. This bytecode can then be executed using the PICOBIT virtual machine. In order to produce as compact as possible bytecode, some specialized optimizations have been added to the compiler. Most of these optimizations are made possible by the extensive use of whole-program analysis throughout the compiler. When compiling a program, PICOBIT appends it to its standard library and compiles the result. By compiling applications and the standard library as a single program, all the whole-program analyzes done in the compiler also apply to the standard library, which leads to more optimization opportunities.

In addition to using selected optimizations to achieve low code sizes, we have designed a custom instruction set, shared by the PICOBIT Scheme compiler and the PICOBIT virtual machine. Much of design of this custom bytecode was geared towards representing common idioms in a compact fashion, with the goal of achieving small application sizes.

6.1 Optimizations

Keeping in mind that the goal of the PICOBIT Scheme system is to produce compact code, the optimizations implemented in the PICOBIT Scheme compiler were chosen mostly for their effect on the resulting code size.

In order to minimize the number of allocations done at runtime, a mutability analysis is done over the whole program at compile-time. Variables that are never mutated are not allocated in memory at runtime, reducing the program's memory footprint and eliminating some variable bookkeeping code, reducing the application code size. For this mutability analysis to be valid, the compiler must analyze the whole program at the same time, which makes PICOBIT's single-program compilation process interesting.

The PICOBIT Scheme compiler also does branch tensioning. Whenever a branch instruction points to another branch instruction, the destination of the first is changed to that of the second, and so on in case of longer branch series. While this optimization is reasonably useful in most compilers, combining it with single-program compilation opens up new possibilities. When using separate compilation, inter-module branches cannot be tensioned, since the nature of such a branch's destination is unknown. However, when using single-program compilation, all destinations are known, and what would have been inter-module branches can be tensioned like any other branches, which leads to more optimization opportunities.

Finally, a treeshaker was added to the PICOBIT Scheme compiler in order to remove any code that is not actually used in the program from the resulting bytecode. A depth-first search is done on the application (and the standard library) to determine which procedures are reachable from the top level. Only these procedures then end up being compiled to bytecode. The rest are simply ignored.

The use of whole-program compilation combined with a treeshaker has an obvious advantage over the use of separate compilation and linking. When using separate compilation, each compilation unit has to be compiled in its entirety, as it is impossible to know before linking which of its procedures will actually be used. With our approach, however, the unreachable code is automatically excluded from the final binary, resulting in smaller application code sizes.

This treeshaker makes it possible to have a well-furnished standard library and still generate compact output, since any unused library procedures will not be present in the resulting bytecode. In our case, the PICOBIT standard library compiles down to 2064 bytes of bytecode, which can be rather large compared to the size of some programs. A PICOBIT program that does not use strings will not include the string functions of the standard library, and will therefore save 508 bytes.

6.2 The PICOBIT Bytecode

Since our goal is to compile applications to small amounts of bytecode, much of the design of the bytecode was geared towards representing common idioms as compactly as possible.

The PICOBIT virtual machine is a stack-based virtual machine. Therefore, pushing values on the data stack is a common operation for the vast majority of the programs it runs. As such, effort was put towards representing pushing instructions in a compact way. This was achieved by having pushing instructions of different lengths, as shown in figure 3. When operands are short enough (typically 4 or 5 bits), short instructions can be used, leading to savings in code size.

To make the most of these short instructions, the shortest value encodings are assigned to frequently used values, as explained in section 7.4. In addition, global variable encodings are assigned in decreasing order of frequency of use, so that the most frequently used global variables are assigned the shortest encodings, and can therefore be used with the short instructions.

In addition to short pushing instructions, PICOBIT also supports short relative addressing instructions. In some frequently occurring cases, such as a goto-if-false whose destination is no more than 15 bytecodes away, instructions fit in a single byte, rather than the three bytes of an absolute addressing instruction. To make the most of these instructions, we use trace scheduling to position the destination code as close to the instructions that reference this destination.

Similarly, the whole instruction set was designed so that instructions that occurred frequently when compiling our set of test applications are represented with shorter encodings than seldom used instructions.

Finally, the PICOBIT bytecode features some high-level instructions. Some common operations, such as creating a closure or copying data from a byte vector to another, are done using a single instruction.

7. The PICOBIT Virtual Machine

The PICOBIT virtual machine is the part of the PICOBIT system that resides on the target microcontroller and interprets the bytecode generated by the PICOBIT Scheme compiler. As such, care was taken to build the virtual machine to be as compact as possible, which means that algorithms and data structures are kept simple throughout the virtual machine. That being said, the PICOBIT virtual machine is a full-featured virtual machine which includes a garbage collector, an implementation of unbound precision integers and support for data structures.

7.1 Environment Representation

The PICOBIT virtual machine being a stack-based virtual machine, environments are represented as stacks. These stacks are themselves represented as PICOBIT lists made of cons cells, allocated in the heap. When looking up a variable in an environment, it is therefore necessary to know its depth in the stack at the current execution point, which can be determined statically.

Such a representation allows us to store multiple environments at the same time, which was invaluable when implementing closures, as seen in section 7.6.

7.2 Automatic Memory Management

The PICOBIT virtual machine is unusual among embedded runtimes in that it features automatic memory management. This allows dynamic languages (such as Scheme) to be run easily on top of it. For this purpose, we use a mark-and-sweep garbage collector. Due to the limited amount of memory available on our target systems, a mark-and-sweep garbage collector is especially interesting as the whole heap can be in use at the same time. By comparison, copying garbage collectors can only use half of the available memory at a given time, thereby cutting the heap size in half and limiting the data size of the applications that can be run on a given chip. Another advantage of a mark-and-sweep garbage collector is that the necessary algorithms are simple, which leads to a compact garbage collector.

The Deutsche-Schorr-Waite algorithm [9] is used in the marking phase, and it really shines in an embedded context. Since this algorithm does not need to use a stack to traverse a tree, no memory needs to be allocated for such a stack. Reserving a portion of the heap for such a stack would not be an interesting option, considering the low amount of available memory to begin with. The use of the Deutsche-Schorr-Waite algorithm therefore allows us to use a larger portion of the microcontroller's memory for our heap, enabling more complex applications to be run using PICOBIT.

`000xxxxx`	Push constant x
`001xxxxx`	Push stack element #x
`0100xxxx`	Push global #x
`0101xxxx`	Set global #x to TOS
`0110xxxx`	Call closure at TOS with x arguments
`0111xxxx`	Jump to closure at TOS with x arguments
`1000xxxx`	Jump to entry point at address $pc + x$
`1001xxxx`	Go to address $pc + x$ if TOS is false
`1010xxxx xxxxxxxx`	Push constant x
`10110000 xxxxxxxx xxxxxxxx`	Call procedure at address x
`10110001 xxxxxxxx xxxxxxxx`	Jump to entry point at address x
`10110010 xxxxxxxx xxxxxxxx`	Go to address x
`10110011 xxxxxxxx xxxxxxxx`	Go to address x if TOS is false
`10110100 xxxxxxxx xxxxxxxx`	Build a closure with entry point x
`10110101 xxxxxxxx`	Call procedure at address $pc + x - 128$
`10110110 xxxxxxxx`	Jump to entry point at address $pc + x - 128$
`10110111 xxxxxxxx`	Go to address $pc + x - 128$
`10111000 xxxxxxxx`	Go to address $pc + x - 128$ if TOS is false
`10111001 xxxxxxxx`	Build a closure with entry point $pc + x - 128$
`10111110 xxxxxxxx`	Push global #x
`10111111 xxxxxxxx`	Set global #x to TOS
`11xxxxxx`	Primitives (`+`, `return`, `get-cont`, ...)

Figure 3. The PICOBIT instruction set and its bytecode encoding

Encoding	PICOBIT	PICOBIT Light
0	`#f`	`#f`
1	`#t`	`#t`
2	`'()`	`'()`
3	-1	-1
4	0	0
5 - 44	1 .. 40	1 .. 40
45 - 127	41 - 123	ROM values
128 - 255	124 - 251	Heap values
256 - 259	252 - 255	N/A
260 - 511	ROM values	N/A
512 - 4095	Heap values	N/A
4096 - 8191	Vector space	N/A

Figure 4. Object encoding in PICOBIT and PICOBIT Light

7.3 Address Space Layout

The distinction between RAM and ROM is important in embedded systems, especially for single-chip microcontrollers. Since there is usually more ROM than RAM available, it is interesting to move as much data as possible to ROM, to leave as much room in RAM as possible for mutable data. Literal values and variables that are never mutated are stored in ROM whereas mutable variables and temporaries are stored in RAM. Therefore, objects manipulated by the PICOBIT virtual machine can be located either in ROM or in RAM.

To reference these objects, the full version of PICOBIT uses 13-bit encodings, whereas the Light version uses 8-bit encodings. Using shorter encodings obviously reduces the number of objects that can be referenced, as shown in figure 4, but since 8-bit encodings can be manipulated using 8-bit rather than 16-bit operations, their use leads to a more compact virtual machine on 8-bit architectures.

In order for objects to contain references to objects stored both in ROM and in RAM, it was necessary to partition PICOBIT's address space. For instance, a pair (whose internal layout is discussed in section 7.4) could have its car stored in ROM and its cdr stored in RAM, in the heap. To reflect this address space partition, the object reference determines whether it points towards a ROM object or a RAM object. As we can see in figure 4, an object encoded with the value 260 is the first object present in ROM, whereas one with value 1285 is the sixth object of the RAM heap.

References can denote ROM and RAM objects, and also preallocated constants that occupy no memory. As shown in figure 4, references with a value from 0 to 259 (0 to 44 for PICOBIT Light) refer to immediate values. Preallocating commonly used values (false, true, the empty list and all numbers that can be represented in a byte) reduces the amount of memory, both ROM and RAM, required to store values. Many common operations, in particular arithmetic on small numbers, can therefore be done without allocating any memory. Furthermore, since special short instructions (see section 6.2) exist to handle references with small values, the use of these frequently occurring preallocated constants can help reduce the size of application bytecode.

Finally, the fourth zone of PICOBIT's address space is used for vectors. The use of this zone will be detailed in section 7.8.

It is worth noting that, to simplify, and therefore reduce the size of, the virtual machine, RAM and ROM objects have the same layout, which only depends on their type, not on their location. More details about these layouts are found in section 7.4.

Another interesting feature of the PICOBIT virtual machine is that its data stack and its continuations are stored as lists in the heap. By storing these in the heap, there is not need to explicitly deallocate stack frames once they are popped or continuations once they are returned from, the garbage collector will automatically deallocate any unreachable objects. The absence of a dedicated allocation scheme for these objects keeps the virtual machine compact, and helps the implementation of closures and first-class continuations, as seen in sections 7.6 and 7.7.

7.4 Object Representation

The PICOBIT virtual machine being designed for dynamic languages, it is necessary to encode objects stored in memory along with their type and garbage collection information.

First of all, all objects are 32 bits wide, whether they are stored in ROM, along with the program, or in RAM, in the heap. We can therefore consider the heap as a simple array of objects, and short indices can be used to refer to objects instead of longer pointers,

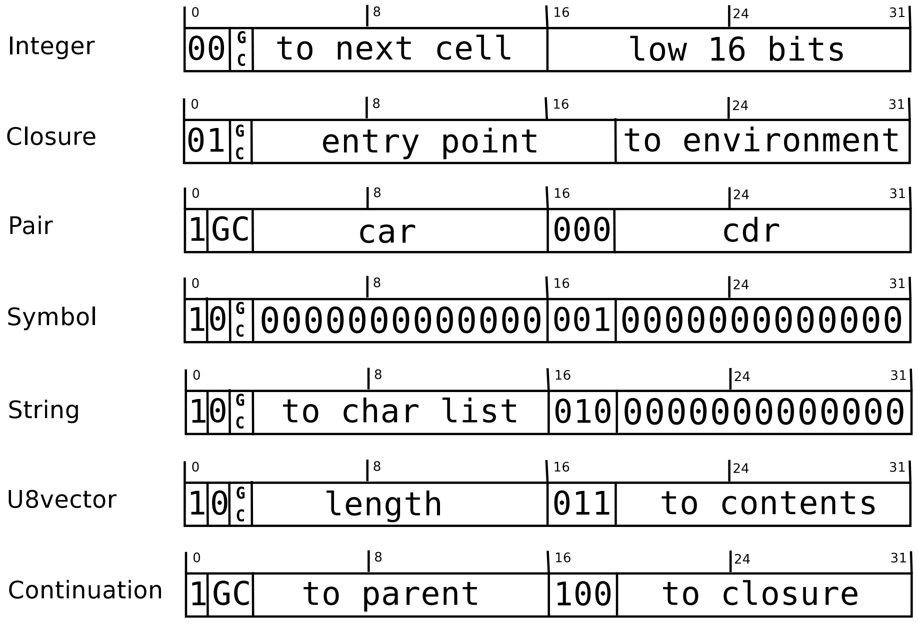
Integer
0 8 16 24 31
00 GC to next cell low 16 bits
Closure
01 GC entry point to environment
Pair
1 GC car 000 cdr
Symbol
1 0 GC 0000000000000 001 0000000000000
String
1 0 GC to char list 010 0000000000000
U8vector
1 0 GC length 011 to contents
Continuation
1 GC to parent 100 to closure

Figure 5. Object encodings in PICOBIT

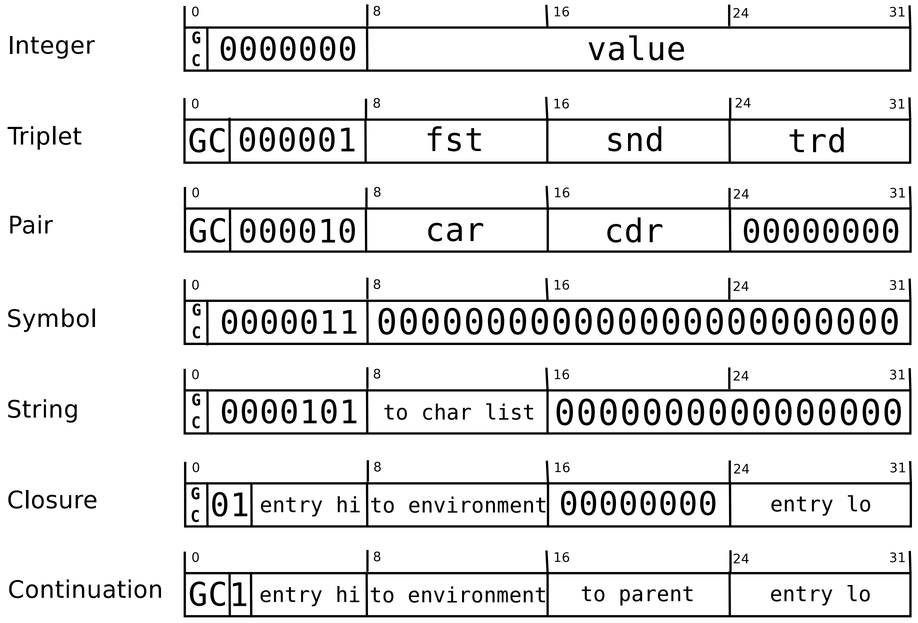
Integer
0 8 16 24 31
GC 0000000 value
Triplet
GC 000001 fst snd trd
Pair
GC 000010 car cdr 00000000
Symbol
GC 0000011 000000000000000000000000
String
GC 0000101 to char list 0000000000000000
Closure
GC 01 entry hi to environment 00000000 entry lo
Continuation
GC 1 entry hi to environment to parent entry lo

Figure 6. Object encodings in PICOBIT Light

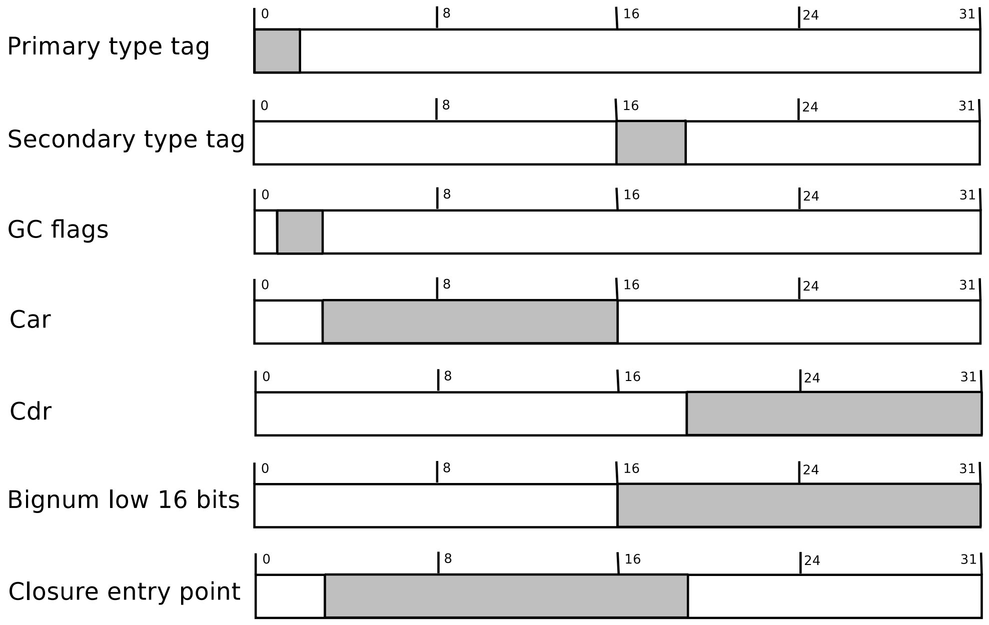

Figure 7. Data accessors in PICOBIT

```
void sweep () {
  obj visit = MAX_HEAP;
  free_list = 0;
  while (visit >= MIN_HEAP) {
    if (marked(visit))
      clear_gc_tags(visit);
    else {
      /* add to the free list */
      set_car (visit, free_list);
      free_list = visit;
    }
    visit--;
  }
}
```

Figure 8. Pseudo-code for the sweeping function of the PICOBIT virtual machine's garbage collector

which leads to a compact object representation. Having a single object size also simplifies garbage collection. Instead of having to figure out where objects begin and end, the sweeping phase of the garbage collector only has to iterate on the array representing the heap. In addition, since the garbage collection flags are located in the same place for objects of all types, it is not necessary to know the exact type of an object when sweeping it. Pseudo-code for the whole sweeping procedure is shown in figure 8.

In addition to being all the same size, PICOBIT objects all follow the same general structure, as shown in figure 5. These similarities reduce the number of virtual machine primitives needed to access the data contained in objects, as the same primitives can be used on most data types, as shown in figure 7. Once again, needing fewer data access primitives helps keeping the PICOBIT virtual machine's size small.

7.5 Unbound Precision Integers

A feature that sets PICOBIT apart from most other embedded programming environments is the availability of unbound precision integers. Traditionally, embedded programming environments on 8-bit microcontrollers offer support for numeric values up to 32 bits wide. However, larger values are needed in some embedded applications. For instance, the S^3 network stack, which runs on top of the PICOBIT system, uses 48 bit integers to store MAC addresses. Large integral values are also necessary for some cryptographic calculations, for instance the SHA [1] family of cryptographic hashing functions, which need values up to 512 bits wide.

Embedded applications also often need to keep track of time, sometimes with a high degree of precision (when controlling machinery, for example). If an application keeps track of time at the microsecond level using a 32-bit value, a wraparound will occur every hour or so. To handle such wraparounds, complex logic might have to be included in the application, leading to an ad-hoc bignum implementation.

Without support for unbound precision integers, these examples could have been implemented using byte vectors instead of large integers, but the necessary calculations would have been awkward and, especially, would require more PICOBIT instructions to encode. For example, the addition of two 16 bit numbers encoded in byte vectors would require an addition for the two low bytes, an overflow check, a second addition for the carry and a third addition for the two high bytes. On the other hand, using unbound precision integers, this requires a single PICOBIT addition instruction. Without even considering problems that might arise if the sum of our two numbers cannot be correctly expressed with 16 bits, we observe that the use of unbound precision integers helps generate compact application bytecode. The size difference become even more noticeable when working with larger values or with multiplications.

As can be seen in figure 5, unbound precision integers are encoded in PICOBIT as linked lists of 16 bit values. At the end of each list is either the integer 0 or -1, which represents the sign. Note that both of these integers have a link to themselves. This simplifies the handling of integers of different lengths and thus the size of the virtual machine. For instance, for addition of two integers, the linked lists are traversed simultaneously until 0 or -1 is reached in both lists.

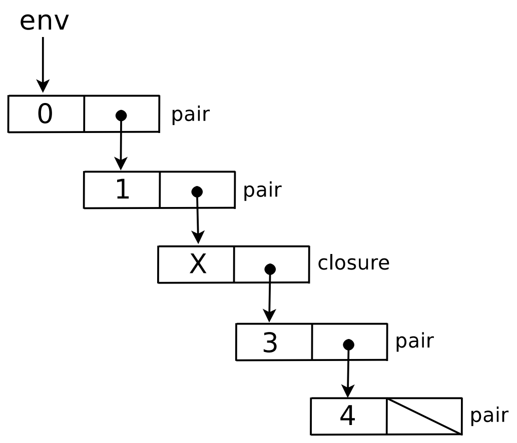

Figure 9. Closed environment look-up

On versions of PICOBIT which do not support unbound precision integers (including PICOBIT Light), integers are limited to 24 bits, and encoded directly in the object, as seen in figure 6.

7.6 Closures

Scheme being a functional language, PICOBIT offers support for closures. Closures are represented as heap objects containing a pointer to the entry point of their associated procedure (in the program space) and a reference to the environment it was created in.

By having the reference to their enclosed environment in the cdr, closures can be traversed as part of a regular environment look-up. Therefore, environment look-up is done in exactly the same way regardless of if the variable is local or closed, the look-up index just has to be adjusted so the look-up skips over the closure in case of a closed variable look-up. This technique works for any number of closures in the environment stack.

7.7 First-Class Continuations

Most Scheme systems implement first-class continuations by copying the stack into the heap with each call to `call/cc`, which can cause an important overhead both in terms of speed and in terms of space.

PICOBIT avoids this overhead by avoiding the use of a call stack, and directly allocating each continuation in the heap like any other object. Manipulating continuations is therefore as simple and efficient as manipulating any other object. In effect, this representation gives us first-class continuations for free. Thus, operations on continuations are implemented as simple virtual machine instructions. Being allocated in the heap, discarded continuations are garbage collected, regardless of how they have been used.

As seen in figure 5, continuations are represented as a chain of continuation objects, each containing a reference to their parent continuation and a reference to a closure object. As explained above, the closure object contains the entry point of the function associated to the continuation and the enclosed environment.

This representation of continuations is very compact, with two objects (the continuation object and the closure object) per frame. When using the multithreading system presented in figure 2, each thread only causes an overhead of one continuation frame, or 8 bytes. Applications with several threads, such as systems monitoring multiple sources of input, can thus be implemented with a very low memory footprint.

7.8 Byte Vectors

Many embedded applications make use of vectors of bytes. For instance, most applications dealing with communication protocols (such as the S^3 network stack) require the use of buffers to store binary data. Byte vectors, mostly because of their constant time random access, are ideal for that use.

Unlike other PICOBIT objects, byte vectors do not necessarily occupy four bytes. In order to guarantee fast random access, byte vectors have to be allocated as a single contiguous space of the appropriate size. To preserve the advantages brought by having all objects of the same size in the heap, we allocate byte vectors in a different section of memory. As such, references with values over 1279 point to objects within this zone, which we call the vector space.

Like the heap, the vector space is allocated by increments of four bytes. However, unlike with the heap, contiguous segments of any length (bounded by the size of the vector space) can be allocated in the vector space. A simple first-fit allocation algorithm is used to decide where to allocate each byte vector.

In addition to the vector contents which are located in the vector space, vectors are also composed of a header, containing the length of the vector and a pointer to the start of the contents (as seen in figure 5). These headers are stored in the heap, and as such are four bytes wide and follow the same general layout as any heap object.

The vector space is not explicitly garbage collected. Instead, when there is not enough space left to allocate a new vector, the garbage collection of the heap is launched, which will likely free some vectors, whose contents' locations are then added to the vector space's free list. Therefore, there is no need for a second garbage collector specifically for the vector space. The absence of a second garbage collector once again helps in keeping the size of the virtual machine low.

PICOBIT Light does not offer support for byte vectors, which removes the need for a separate vector space, and simplifies several algorithms of the virtual machine, leading to a more compact VM.

8. The SIXPIC C Compiler

When using the PICOBIT Scheme system, the total size of the software running on the target system is the sum of the size of the PICOBIT virtual machine and of the application programs, and out goal is to minimize that total size. As we have seen earlier, the PICOBIT Scheme compiler was designed to generate compact application bytecode. That leaves the size of the virtual machine, which, in some cases, can account for an important part of the whole system. While the PICOBIT virtual machine can be compiled with any C compiler, some savings in code size can be achieved by using a specialized C compiler to compile it. The SIXPIC C compiler is one such compiler.

The SIXPIC C compiler was designed to generate compact code, especially when compiling virtual machines. This was done by analyzing the code of typical virtual machines (including the PICOBIT virtual machine) to find and then optimize common patterns found within them. This analysis also showed us which features of the C language were seldom used for virtual machines, and could therefore be omitted from SIXPIC. In addition to reducing the complexity of the compiler, some of these omissions also opened possibilities for optimization which would not have been valid otherwise.

The SIXPIC C compiler currently compiles C to machine code for Microchip's PIC18 family of microcontrollers, which is commonly used in small embedded systems. However, the techniques explained here are not PIC18-specific, and would be useful on most microcontroller architectures. In fact, most of the SIXPIC C compiler's compilation process is architecture-independent and could be used for most 8-bit microcontroller families.

8.1 Restrictions

Even though virtual machines are complex pieces of software, they do not make use of every single feature of the C language. Therefore, while designing SIXPIC, some features could be left out and some others were restricted to the subset actually used by most virtual machines.

The first notable omission is that of support for floating point numbers. Since the PICOBIT Scheme system does not support them, and that most microcontrollers do not support floating point operations, this omission is pretty straightforward.

Since virtual machines typically manage their data structures at the bit level (especially in embedded systems), ordinary C structs are not generally useful in the context of virtual machines.

A more controversial restriction would be that SIXPIC does not support recursive (or mutually-recursive) functions. At first glance, this might appear restrictive. However, since typical virtual machines consist mostly of a `switch` statement in a loop, recursion is not usually needed. This omission is what makes our specialized calling convention (see subsection 8.2) possible.

Finally, arrays in SIXPIC are restricted to byte arrays. Since the SIXPIC C compiler targets the PIC18 family of microcontrollers, which are 8-bit systems, the performance of arrays of larger values would have been much worse than that of byte arrays. While this might seem limitating at first, especially considering that PICOBIT objects are 32 bits wide, it is not the case. Most of the time, data stored in objects is read only one field at a time, most of the time only the header. As shown in figure 7, all fields of PICOBIT objects (with the exception of closure entry points) span at most two bytes, and the header always fits in one byte. While having to make two array accesses to read a single field might seem slow, it is not much slower than a single access to an array of 16 bit-wide values on an 8-bit system.

8.2 Calling Convention

To support recursive functions, a call stack is usually needed. Most modern workstation architectures provide hardware support for such stacks, which makes the compiler's job easier. However, most microcontroller architectures do not offer such support, which means that the compiler would need to build a software stack in memory in order to support recursive functions. The building and use of such a stack increases the complexity, and therefore the size, of the generated code. By giving up support for recursive functions, no such stack is needed anymore and it becomes possible to use a calling convention which passes function arguments in pre-determined registers. This approach is taken by the leading embedded C compilers, such as Microchip's Hi-Tech C© compiler.

With the SIXPIC C compiler, we take this approach further. Since we do not support recursive functions, every variable (be it a local variable, a global variable, or a function parameter) can be allocated at a static location. We then use whole-program analysis to determine which variables interfere with each other and use the results to do register allocation for the whole program all at once.

Since the location of each variable is known at compile-time, we can avoid moving values to and from the registers needed by the calling convention. Instead, we use a specialized calling convention where the caller moves the arguments directly in the registers where the callee's local variables reside, as shown on figure 10.

The analyzes we use for this whole-program register allocation, liveness analysis and interference graph construction, are typically used as intra-procedural analyzes. However, when using them globally, the traditional algorithms scale poorly, resulting in slowdowns.

```
byte f (byte x) {
  return x + 3;
}
...
byte y = 3;
f(y);
```

Stack-based calling convention:

```
...
   push $y
   call $f
...
f: pop $x
...
```

Total: 20 bytes of PIC18 machine code.

Register-based calling convention:

```
...
   move $y A ; from y to the predetermined register
   call  $f
...
f: move A $x ; from the predetermined register to x
...
```

Total: 12 bytes of PIC18 machine code.

Specialized calling convention:

```
...
   move $y $x ; directly from y to x
   call  $f
...
f:
...
```

Total: 8 bytes of PIC18 machine code.

Figure 10. Comparison between a stack-based calling convention, a register-based calling convention and our specialized calling convention

To avoid these slowdowns, we developed variations of these algorithms that ended up being more suitable to use as global analyzes.

Of course, since this calling convention is dependent on whole-program analysis, the use of external libraries is not as transparent as with other C compilers. To use external libraries with the SIXPIC C compiler, it is necessary to compile the libraries and the original program as a single program, which means our specialized calling convention will be used through the whole program, leading to more compact code.

8.3 Optimizations

As with the PICOBIT Scheme compiler, the optimizations present in the SIXPIC C compiler were chosen for their impact in reducing the size of the resulting code.

First of all, our register allocation algorithm does register coalescing. Since the SIXPIC C compiler does whole-program register allocation, register coalescing can be used more broadly. Instead of being limited to coalescing virtual registers inside each function, as would be the case with intra-procedural register allocation, global register allocation makes it possible to coalesce registers being used in two different functions. With our specialized calling convention (see subsection 8.2), such opportunities occur enough to be worthwhile. We measured that the use of register coalescing reduces the size of the generated code by around 4.5%, mostly by eliminating move instructions between coalesced registers. Out of the 2420 byte cells found in the PICOBIT virtual machine, 1453 end up being coalesced. After register allocation, only 324 bytes of RAM are necessary to run the VM.

Since virtual machines traditionally do a lot of accesses to a restricted number of arrays (the heap and the program space, for instance), SIXPIC provides a way to flag two byte arrays as being more important than others. This has the effect of allocating pointers to these arrays directly in the PIC18 array registers instead of allocating them in regular memory. Since the PIC18 (and most 8-bit microcontroller architectures) lack sophisticated addressing modes, this optimization helps reduce the number of operations needed for array accesses.

By looking for patterns in the code of several virtual machines, we noticed that the `switch/case` construct was extensively used, especially for instruction decoding. PICOBIT is no exception. We also noticed that most of the `switch/case` statements used in virtual machines respected several other properties, including the absence of `default` labels and the presence of mostly contiguous label numbers. We therefore worked on an implementation of `switch/case` that would generate compact code, especially when the above properties hold. After trying several implementations, we settled on a branch table-based approach which, despite the absence of computed branches on the PIC18 architecture, generates compact code in the cases that interest us.

Like the PICOBIT Scheme compiler, SIXPIC does trace scheduling. The benefits explained in section 6.1 also apply to SIXPIC, since it also does single-program compilation. When compiling the PICOBIT virtual machine, 519 jumps are shortened thanks to trace scheduling and 228 are eliminated altogether, which saves 6.3% of the virtual machine size.

Instead of providing an external standard library with which applications can be linked, the SIXPIC C compiler's standard library is defined in terms of the compiler's intermediate representation (control flow graphs with abstract 3-argument instructions). When compiling a program, SIXPIC joins the standard library's control flow graph to the program's, and uses the resulting graph for the rest of the compilation process. Therefore, all the whole-program optimizations describes above are run on the standard library at the same time, resulting in a greater optimization potential.

Finally, the SIXPIC C compiler uses, like the PICOBIT Scheme compiler, a treeshaker to remove any unused code from the generated executable, reducing its size. As is the case with the Scheme compiler, SIXPIC appends its standard library to application programs, then compiles only the reachable parts. Once again, the use of this treeshaker helps SIXPIC achieve low application code sizes by excluding unused code in the application (or in the standard library).

9. Experimental Results

9.1 Bytecode-Based Approach

As we anticipated, a bytecode-based approach to embedded application development leads to compact application sizes.

On figure 11, we show examples of programs used with the PICOBOARD robot, and the amount of bytecode required for each. As we can see, all these programs, even relatively sophisticated ones like a web server, can be represented compactly using bytecode. It is worth noting that these small code sizes were obtained despite PICOBIT having a large (2064 bytes) standard library, thanks to the treeshaker, which removes unused parts of the library from the final bytecode.

Program	Code size (B)
Flashing led	9
Follow the light	101
Remote control	106
Hello	355
Light sensors	374
Multi-threaded presence counter	599
Web server	1033

Figure 11. Example PICOBOARD programs

Stack	Code size (kB)	VM size (kB)	Total size (kB)
S^3	3.1	15.6	18.7
uIP	10.0	-	10.0

Figure 12. Comparison between the S^3 and uIP embedded network stacks

We have also compared the S^3 TCP/IP stack, which is used with the PICOBIT Scheme system, to Adam Dunkels's uIP [4] stack, which is written in C and is compiled natively to machine code. Both stacks implement a similar set of features and share most design decisions. They can be therefore considered roughly equivalent for our comparison's purposes.

When compiling S^3 with the PICOBIT Scheme compiler, we obtain 3.1 kB of bytecode whereas when we compile the uIP stack using Microchip's MCC18 compiler, we obtain a 10.0 kB binary. We notice that compiling to bytecode resulted in the application being about three times as compact.

Since the bytecode is useless without the PICOBIT virtual machine to interpret it, we have to include the size of the virtual machine to get realistic figures. When comparing the size of the whole systems (see figure 12), the natively compiled uIP is about twice as compact as the combination of S^3 and of the PICOBIT virtual machine.

However, it is worth noting that the size of the virtual machine is a fixed cost which is independent of the size of the application it interprets. Therefore, the cost of the virtual machine is amortized over all the applications it executes.

Since TCP/IP stacks are complex applications, we believe that the compactness of bytecode versus machine code that we have observed when compiling S^3 would hold when compiling other complex applications. We therefore expect that for sufficiently large applications, our bytecode-based approach would lead to smaller system sizes than a native compilation-based one. Due to their smaller size, we expect that the restricted versions of PICOBIT will fare even better in this regard.

Keeping in mind that our motivation was to execute larger programs on smaller chips, the fact that our bytecode-based approach will likely behave better than native compilation for sufficiently large programs is promising.

9.2 Specialized C compiler

Another key element of our approach towards embedded development is the use of a specialized C compiler optimized towards virtual machines. So far, this approach looks promising, but a sufficiently optimizing general-purpose C compiler can still generate more compact code than our specialized SIXPIC C compiler, as is shown in figure 13.

Thanks to its domain-specific optimizations, SIXPIC outperforms Microchip's MCC18 general-purpose C compiler by about 42% when compiling the PICOBIT virtual machine. However the more mature Microchip's Hi-Tech C compiler generates code that is 12% more compact than SIXPIC's, likely due to its broader

Version	SIXPIC	MCC18	Hi-Tech C
Full PICOBIT	17.5 kB	24.8 kB	15.6 kB
Without bignums	13.0 kB	17.0 kB	11.6 kB
PICOBIT Light	7.2 kB	8.0 kB	5.2 kB

Figure 13. Size comparison between the different versions of the PICOBIT VM compiled with various C compilers

range of general-purpose optimizations. We expect that adding more domain-specific optimizations to the SIXPIC C compiler will allow it to close the gap.

10. Future Work

While some work has already been done towards make the PICOBIT bytecode compact, it has mostly consisted in observing the generated code and finding more compact encodings by hand. An interesting, and more rigorous, approach would be to use Huffman encoding on the bytecode to further reduce its size. Such an approach has been successful [8] for several virtual machines, and could lead to reductions in application code size.

Some work also remains to be done on the SIXPIC C compiler to handle in a more compact fashion some common virtual machine idioms. So far, work has been done to leverage several interesting properties of virtual machines, most notably their lack of recursive functions, but some observed virtual machine patterns are not yet properly exploited by SIXPIC.

Finally, as previously mentioned, the instruction set and data types of the PICOBIT virtual machine, even though they were chosen and designed with the Scheme language in mind, are general enough to support other dynamic languages, such as Python or Perl. The Factor language also comes to mind, as a dynamically-typed garbage-collected stack-based language could integrate well with the stack-based PICOBIT virtual machine.

11. Conclusion

We have presented an implementation of the Scheme programming language which is suitable for programming small microcontrollers. The system supports several high-level constructs not usually available in microcontroller development tools, including garbage collection, higher-order procedures, first-class continuations, threads, and unbound precision integers. Our approach tackles the space issue in three ways: the design of a Scheme compiler generating compact bytecode, a small virtual machine, and an optimizing C compiler suited to the compilation of the virtual machine. Although there are still avenues for improvement that we will pursue in our future work, our results show that a fairly featurefull Scheme system can run on platforms with only a few kilobytes of memory. For instance, it allows a basic network protocol stack (S^3) to run on a microcontroller with less than 19 kB of ROM.

References

[1] *Secure Hash Standard*. National Institute of Standards and Technology, Washington, 2002. Federal Information Processing Standard 180-2.

[2] W. Clinger and J. Rees. The revised4 report on the algorithmic language scheme, Nov 1991.

[3] D. Dube. BIT: A very compact scheme system for embedded applications. In *Proceedings of the Workshop on Scheme and functional programming*, 2000.

[4] A. Dunkels. Full TCP/IP for 8-bit architectures. In *MobiSys '03: Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 85–98, New York, NY, USA, 2003. ACM.

[5] M. Feeley. A better API for first-class continuations. In *2nd Workshop on Scheme and Functional Programming*, 2001.

[6] M. Feeley and D. Dube. PICBIT: A Scheme system for the PIC microcontroller. In *Proceedings of the Fourth Workshop on Scheme and Functional Programming*, 2003.

[7] R. Kelsey, W. Clinger, and J. Rees. The Revised^5 Report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1), Sep 1998.

[8] M. Latendresse and M. Feeley. Generation of fast interpreters for huffman compressed bytecode. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 32–40, New York, NY, USA, 2003. ACM.

[9] 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.

[10] V. St-Amour, L. Bouchard, and M. Feeley. Small Scheme Stack: a Scheme TCP/IP stack targeting small embedded applications. In *2008 Workshop on Scheme and Functional Programming*, 2008.

[11] I. Sun Microsystems. Java card 3.0.1 platform specification, May 2009.