PICBIT: A Scheme System for the PIC Microcontroller

Marc Feeley / Université de Montréal
Danny Dubé / Université Laval

ABSTRACT

This paper explains the design of the PICBIT R^{4}RS Scheme system which specifically targets the PIC microcontroller family. The PIC is a popular inexpensive single-chip microcontroller for very compact embedded systems that has a ROM on the chip and a very small RAM. The main challenge is fitting the Scheme heap in only 2 kilobytes of RAM while still allowing useful applications to be run. PICBIT uses a novel compact (24 bit) object representation suited for such an environment and an optimizing compiler and byte-code interpreter that uses RAM frugally. Some experimental measurements are provided to assess the performance of the system.

1 INTRODUCTION

The Scheme programming language is a small yet powerful high-level programming language. This makes it appealing for applications that require sophisticated processing in a small package, for example mobile robot navigation software and remote sensors.

There are several implementations of Scheme that require a small memory footprint relative to the total memory of their target execution environment. A full-featured Scheme system with an extended library on a workstation may require from one to ten megabytes of memory to run a simple program (for instance MzScheme v205 on Linux has a 2.3 megabyte footprint). At the other extreme, the BIT system [1] which was designed for microcontroller applications requires 22 kilobytes of memory on the 68HC11 microcontroller for a simple program with the complete R^{4}RS library (minus file I/O). This paper describes a new system, PICBIT, which is inspired from our BIT system and specifically designed for the PIC microcontroller family which has even tighter memory constraints.

Permission to make digital or hard copies, to republish, to post on servers or to redistribute to lists all or part of this work is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To otherwise copy or redistribute requires prior specific permission. Fourth Workshop on Scheme and Functional Programming. November 7, 2003, Boston, Massachusetts, USA. Copyright 2003 Marc Feeley and Danny Dubé.

2 THE PIC MICROCONTROLLER

The PIC is one of the most popular single-chip microcontroller families for low-power very-compact embedded systems [6]. There is a wide range of models available offering RISC-like instruction sets of 3 different complexities (12, 14, or 16 bit wide instructions), chip sizes, number of I/O pins, execution speed, on-chip memory and price. Table 1 lists the characteristics of a few models from the smallest to the largest currently available.

BIT was originally designed for embedded platforms with 10 to 30 kilobytes of total memory. We did not distinguish read-only (ROM) and read-write (RAM) memory, so it was equally important to have a compact object representation, a compact program encoding and a compact runtime. Moreover the design of the byte-code interpreter and libraries favors compactness of code over execution speed, which is a problem for some control applications requiring more computational power. The limited range of integers (-16384 to 16383) is also awkward. Finally, the incremental garbage collector used in BIT causes a further slowdown in order to meet real-time execution constraints [2].

Due to the extremely small RAM of the PIC, it is necessary to distinguish what needs to go in RAM and what can go in ROM. Table 1 shows that for the PIC there is an order of magnitude more ROM than RAM. This means that the compactness of the object representation must be the primary objective. The compactness of the program encoding and runtime is much less of an issue, and can be traded-off for a more compact object representation and speedier byte-code interpreter. Finally, we think it is probably acceptable to use a nonincremental garbage collector, even for soft real-time applications, because the heap is so small.

We call our Scheme system PICBIT to stress that the characteristics of the PIC were taken into account in its design. However the system is implemented in C and it should be easy to port to other microcontrollers with similar memory constraints. We chose to target the "larger" PIC models with 2 kilobytes of RAM or more (such as the PIC18F6520) because we believed that this was the smallest RAM for doing useful work. Our aim was to create a practical system

Model	Pins	MIPS	ROM	RAM	Price
PIC12C508	8	1	512 × 12 bits	25 × 8 bits	$0.90
PIC16F628	18	5	2048 × 14 bits	224 × 8 bits	$2.00
PIC18F6520	64	10	16384 × 16 bits	2048 × 8 bits	$6.50
PIC18F6720	64	6.25	65536 × 16 bits	3840 × 8 bits	$10.82

Table 1: Sample PIC microcontroller models.

that strikes a reasonable compromise between the conflicting goals of fast execution, compact programs and compact object representation.

3 OBJECT REPRESENTATION

3.1 Word Encoding

In many implementations of dynamically-typed languages all object references are encoded using words of W bits, where W is often the size of the machine's words or addresses [3]. With this approach at most 2^W references can be encoded and consequently at most 2^W objects can live at any time. Each object has its unique encoding. Since many types of objects contain object references, W also affects the size of objects and consequently the number of objects that can fit in the available memory. In principle, if the memory size and mix of live objects are known in advance, there is an optimal value for W that maximizes the number of objects that can coexist.

The 2^W object encodings can be partitioned, either statically (e.g. tag bits, encoding ranges, type tables) or dynamically (e.g. BIBOP [4]) or a combination, to map them to a particular type and representation. A representation is `direct` if the W bit word contains all the information associated with the object, e.g. a fixnum or Boolean (the meaning of "all the information" is left vague). In an `indirect` representation the W bit word contains the address in memory (or an index in a table) where auxiliary information associated with the object is stored, e.g. the fields of a pair or string. The direct representation can't be used for mutable objects because mutation must only change the state of the object, not its identity. When an indirect representation is used for immutable objects the auxiliary information can be stored in ROM because it is never modified, e.g. strings and numbers appearing as literals in the program.

Like many microcontrollers, the PIC does not use the same instructions for dereferencing a pointer to a RAM location and to a ROM location. This means that when the byte-code interpreter accesses an object it must distinguish with run time tests objects allocated in RAM and in ROM. Consequently there is no real speed penalty caused by using a different representation for RAM and ROM, and there are possibly some gains in space and time for immutable objects.

Because the PIC's ROM is relatively large and we expect the total number of immutable objects to be limited, using the indirect representation for immutable objects requires relatively little ROM space. Doing so has the advantage that we can avoid using some bits in references as tags. It means that we do not have to reserve in advance many of the 2^W object encodings for objects, such as fixnums and characters, that may never be needed by the program. The handling of integers is also simplified because there is no small vs. large distinction between integers. It is possible however that programs which manipulate many integers and/or characters will use more RAM space if these objects are not preallocated in ROM. Any integer and character resulting from a computation that was not preallocated in ROM will have to be allocated in RAM and multiple copies might coexist. Interning these objects is not an interesting approach because the required tables would consume precious RAM space or an expensive sweep of the heap would be needed. To lessen the problem, a small range of integers can be preallocated in ROM (for example all the encodings that are "unused" after the compiler has assigned encodings to all the program literals and the maximum number of RAM objects).

3.2 Choice of Word and Object Size

For PICBIT we decided that to get simple and time-efficient byte-code interpreter and garbage collector all objects in RAM had to be the same size and that this size had to be a multiple of 8 bits (the PIC cannot easily access bit fields). Variable size objects would either cause fragmentation of the RAM, which is to be avoided due to its small size, or require a compacting garbage collector, which are either space- or time-inefficient when compared to the mark-sweep algorithm that can be used with same size objects. We considered using 24 bits and 32 bits per object in RAM, which means no more than 682 and 512 objects respectively can fit in a 2 kilobyte RAM (the actual number is less because the RAM must also store the global variables, C stack, and possibly other internal tables needed by the runtime). Since some encodings are needed for objects in ROM, W must be at least 10, to fully use the RAM, and no more than 12 or 16, to fit two object references in an object (to represent pairs).

With $W = 10$, a 32 bit object could contain three object references. This is an appealing proposition for compactly representing linked data structures such as binary search tree nodes, special association lists and continuations that the interpreter might use profitably. Unfortunately many bits would go unused for pairs, which are a fairly common data type. Moreover, $W = 10$ leaves only a few hundred encodings for objects in ROM. This would preclude running programs that

1. contain too many constant data-structures (the system would run out of encodings);

2. maintain tables of integers (integers would fill the RAM).

But these are the kind of programs that seem likely for microcontroller applications (think for example of byte buffers, state transition tables, and navigation data). We decided

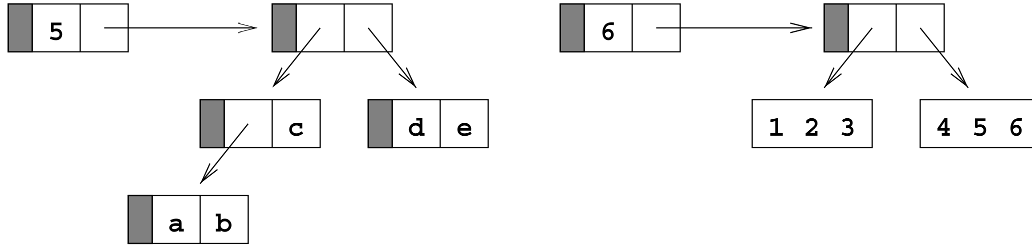

Figure 1: Object representation of the vector `#(a b c d e)` and the string `"123456"`. To improve readability some of the details have been omitted, for example the "a" is really the object encoding of the symbol `a`. The gray area corresponds to the two tag bits. Note that string leaves use all of the 24 bits to store 3 characters.

that 24 bit objects and $W = 11$ was a more forgiving compromise, leaving at least 1366 ($2^W - 682$) encodings for ROM objects.

It would be interesting to perform an experiment for every combination of design choice. However, creating a working implementation for one combination requires considerable effort. Moreover, it is far from obvious how we could automate the creation of working implementations for various spaces of design choice. There are complex interactions between the representation of objects, the code that implements the operations on the objects, the GC, and the parts of the compiler that are dependent on these design choices.

3.3 Representation Details

For simplicity and because we think ROM usage is not an important concern for the PIC, we did not choose to represent RAM and ROM objects differently.

All objects are represented indirectly. That is, they all are allocated in the heap (RAM or ROM) and they are accessed through pointers. Objects are divided in three fields: a two bit tag, which is used to encode type information, and two 11 bit fields. No type information is put on the references to objects. The purpose of each of the two 11 bit fields (X and Y) depends on the type:

00 $\Rightarrow$ Pair. X and Y are object references for the `car` and `cdr`.

01 $\Rightarrow$ Symbol. X and Y are object references for the name of the symbol (a string) and the next symbol in the symbol table. Note that the name is not necessary in a program that does not convert between strings and symbols. PICBIT does not currently perform this optimization.

10 $\Rightarrow$ Procedure. X is used to distinguish the three types of procedures based on the constant C (number of lambdas in the program) which is determined by the compiler, and the constant P (number of Scheme primitive procedures provided by the runtime, such as `cons` and `null?`, but not `append` and `map` which are defined in the library, a Scheme source file):

$0 \leq X < C \Rightarrow$ Closure. X is the entry point of the procedure (raw integer) and Y is an object reference to the environment (the set of nonglobal free variables, represented with an improper list).

$C \leq X < C + P \Rightarrow$ Primitive. X is the entry point of the procedure (raw integer) and Y is irrelevant.

$X = C + P \Rightarrow$ Reified continuation. X is an object reference to a continuation object and Y is irrelevant. A continuation object is a special improper list of the form (r p . e), where r is the return address (raw integer), p is the parent continuation object and e is an improper list environment containing the continuation's live free variables.

The runtime and P are never modified even when some primitive procedures are not needed by the compiled program.

11 $\Rightarrow$ One of vector, string, integer, character, Boolean or empty list. X is a raw integer that determines the specific type. For integer, character, Boolean and empty list, X is less than 36 and Y is also a raw integer. For the integer type, 5 bits from X and 11 from Y combine to form a 16 bit signed integer value. For the vector type, $36 \leq X < 1024$ and $X - 36$ is the vector's length. For the string type, $1024 \leq X < 2048$. To allow a logarithmic time access to the elements of vectors and strings, Y is an object reference to a balanced tree of the elements. A special case for small vectors (length 0 and 1) and small strings (length 0, 1, and 2) stores the elements directly in Y (and possibly 5 bits of X for strings of length 2). Figure 1 gives an example of how vectors and strings are represented. Note that the leaves of strings pack 3 characters.

4 GARBAGE COLLECTION

The mark-sweep collector we implemented uses the Deutsch-Schorr-Waite marking algorithm [7]. This algorithm can traverse a linked data structure without using an auxiliary stack, by reversing the links as it traverses the data structure (we call such reversed links "back pointers"). Conceptually two bits of state are attached to each node. The `mark` bit indicates that the node has been visited. The `stage` bit indicates which of the two links has been reversed.[1] When the

[1] In fact, a `trit` should be attached to each node instead of two bits since the stage bit is meaningless when the mark bit is not set.

marking algorithm returns to a node as part of its backtracking process using the current back pointer (i.e. the "top of the stack"), it uses the stage bit to know which of the two fields contains the next back pointer. The content of this field must be restored to its original value, and if it is the first field then the second field must be processed in turn.

These bits of information cannot be stored explicitly in the nodes because all 24 bits are used. The mark bit is instead stored in a bit vector elsewhere in the RAM (this means the maximal number of objects in a 2 kilobyte RAM is really 655, leaving 1393 encodings for ROM objects).

We use the following trick for implementing the stage bit. The address in the back pointer has been shifted left by one position and the least significant bit is used to indicate which field in the "parent" object is currently reversed. This approach works for the following reason. Note that stage bits are only needed for nodes that are part of the chain of reversed links. Since there are more ROM encodings than RAM encodings and a back pointer can only point to RAM, we can use a bit of the back pointer to store the stage bit. A back pointer contains the stage bit of the node that it points to.

One complication is the traversal of the nodes that don't follow the uniform layout (with two tag bits), such as the leaves of strings that contain raw integers. Note that references to these nodes only occur in a specific type of "enclosing" object. This is an invariant that is preserved by the runtime system. It is thus possible to track this information during the marking phase because the only way to reach an object is by going through that specific type of enclosing object. For example, the GC knows that it has reached the leaf of a string because the node that refers to it is an internal string tree node just above the leaves (this information is contained in the type bits of that node).

After the marking phase, the whole heap is scanned to link the unmarked nodes into the free list. Allocation removes one node at a time from the free list, and the GC process is repeated when the free list is exhausted.

5 BYTE-CODE INTERPRETER

The BIT system's byte-code interpreter is relatively slow compared to other Scheme interpreters on the same platform. One important contributor to this poor performance is the management of intermediate results. The evaluation "stack" where intermediate results are saved is actually implemented with a list and every evaluation, including that of constants and variables, requires the allocation of a pair to link it to the stack. This puts a lot of pressure on the garbage collector, which is not particularly efficient because it is incremental. Moreover, continuations are not safe-for-space.

To avoid these problems and introduce more opportunities for optimization by the compiler, we designed a register-based virtual machine for PICBIT. Registers can be used to store intermediate results and to pass arguments to procedures. It is only when these registers are insufficient that values must be saved on an evaluation stack. We still use a linked representation for the stack, because reserving a contiguous section of RAM for this purpose would either be wasteful (stack section too large) or risk stack overflows (stack section too small). Note that we don't have the option of growing the stack and heap toward each other, because our garbage collector does not compact the heap. Substantial changes to the object representation would be needed to permit compaction.

The virtual machine has six registers containing object references: `Acc`, `Arg1`, `Arg2`, `Arg3`, `Env`, and `Cont`. `Acc` is a general purpose accumulator, and it contains the result when returning to a continuation. `Arg1`, `Arg2`, and `Arg3` are general purpose and also used for passing arguments to procedures. If there are more than three arguments, `Arg3` contains a list of the third argument and above. `Env` contains the current environment (represented as an improper list). `Cont` contains a reference to a continuation object (which as explained above contains a return address, a reference to the parent continuation object and an environment containing the continuation's live free variables). There are also the registers `PC` (program counter) and `NbArgs` (number of arguments) that hold raw integers. When calling an inlined primitive procedure (such as `cons` and `null?`, but not `apply`), all registers except `Acc` and `PC` are unchanged by the call. For other calls, all registers are caller-save except for `Cont` which is callee-save.

Most virtual machine instructions have register operands (source and/or destination). Below is a brief list of the instructions to give an idea of the virtual machine's size and capabilities. We do not explain all the instruction variants in detail.

`CST` *addr*, r ⇒ Load a constant into register r.

`MOV[S]` r_1, r_2 ⇒ Store r_1 into r_2.

`REF(G|[T][B])` i, r ⇒ Read the global or lexical variable at position i and store it into r.

`SET(G|[T][B])` r, i ⇒ Store r into the global or lexical variable at position i.

`PUSH` r_1, r_2 ⇒ Construct the pair `(cons` r_1 r_2`)` and store it into r_2.

`POP` r_1[, r_2] ⇒ Store `(car` r_1`)` into r_2 and store `(cdr` r_1`)` into r_1.

`RECV[T]` n ⇒ Construct the environment of a procedure with n parameters and store it into `Env`. This is normally the first instruction of a procedure.

`MEM[T][B]` r ⇒ Construct the pair `(cons` r_1 `Env)` and store it into `Env`.

`DROP` n ⇒ Remove the n first pairs of the environment in `Env`.

`CLOS` n ⇒ Construct a closure from n (entry point) and `Acc` and store it into `Acc`.

`CALL` n ⇒ Set `NbArgs` to n and invoke the procedure in `Acc`. Register `Cont` is not modified (the instruction does not construct a new continuation).

`PRIM` i ⇒ Inline call to primitive procedure i.

`RET` ⇒ Return to the continuation in `Cont`.

`JUMPF` r, *addr* $\Rightarrow$ If r is false, branch to address *addr*.

`JUMP` *addr* $\Rightarrow$ Branch to address *addr*.

`SAVE` n $\Rightarrow$ Construct a continuation from n (return point), `Cont`, and `Env` and store it into `Cont`.

`END` $\Rightarrow$ Terminate the execution of the virtual machine.

6 COMPILER

PICBIT's general compilation approach is similar to the one used in the BIT compiler. A whole-program analysis of the program combined with the Scheme library is performed and then the compiler generates a pair of C files ("`.c`" and "`.h`"). These files must be compiled along with PICBIT's runtime system (written in C) in a single C compilation so that some of the data-representation constants defined in the "`.h`" file can specialize the runtime for this program (i.e. the encoding range for RAM objects, constant closures, etc). The "`.h`" file also defines initialized tables containing the program's byte-code, constants, etc.

PICBIT's analyses, transformations and code generation are different from BIT's. In particular:

- The compiler eliminates useless variables. Both lexical and global variables are subject to elimination. Normally, useless variables are rare in programs. However, the compiler performs some transformations that turn many variables into useless ones. Namely, constant propagation and copy propagation, which replace references to variables that happen to be bound to constants and to the value of immutable variables, respectively. Variables that are not read and that are not set unsafely (e.g. mutating a yet undefined global variable) are deemed useless.

- Programs typically contain literal constant values. The compiler also handles closures with no nonglobal free variables as constants (this is possible because there is a single instance of the global environment). Note that all library procedures and typically most or all top-level user procedures can be treated like constants. This way globally defined procedures can be propagated by the compiler's transformations, often eliminating the need for the global variable they are bound to.

- The compiler eliminates dead code. This is important, because the R^4RS runtime library is appended to the program and the compiler must try to discard all the library procedures that are unnecessary. This also eliminates constants that are unnecessary, which avoids wasting object encodings. The dead code elimination is based on a rather simplistic test: the value of a variable that is read for a reason other than being copied into a global variable is considered to be required. In practice, the test has proved to be precise enough.

- The compiler determines which variables are live at return points, so that only those variables are saved in the continuations created. Similarly, the environments stored into closures only include variables that are needed by the body of the closures. This makes continuations and closures safe-for-space. It is particularly important for an embedded system to be safe-for-space. For example, an innocent-looking program such as the one in Figure 2 retains a considerable amount of data if the closures it generates include unnecessary variables. PICBIT has no problem executing it.

```
(define (make-list n x)
  (if (<= n 0)
      '()
      (cons x (make-list (- n 1) x))))

(define (f lst)
  (let* ((len (length lst))
         (g (lambda () len)))
    (make-list 100 g)))

(define (many-f n lst)
  (if (<= n 0)
      lst
      (many-f (- n 1) (f lst))))

(many-f 20000 (make-list 100 #f))
```

Figure 2: Program that requires the safe-for-space property.

7 EXPERIMENTAL RESULTS

To evaluate performance we use a set of six Scheme programs that were used in our previous work on BIT.

`empty` Empty program.

`thread` Small multi-threaded program that manages 3 concurrent threads with `call/cc`.

`photovore` Mobile robot control program that guides the robot towards a source of light.

`all` Program which references each Scheme library procedure once. The implementation of the Scheme library is 737 lines of Scheme code.

`earley` Earley's parser, parsing using an ambiguous grammar.

`interp` An interpreter for a Scheme subset running code to sort a list of six strings.

The `photovore` program is a realistic robotics program with soft real-time requirements that was developed for the LEGO MINDSTORMS version of BIT. The source code is given in Figure 3. The other programs are useful to determine the minimal space requirements (`empty`), the space requirements for the complete Scheme library (`all`), the space requirements for a large program (`earley` and `interp`), and to check if multi-threading implemented with `call/cc` is feasible (`thread`).

We consider `earley` and `interp` to be complex applications that are atypical for microcontrollers. Frequently, microcontroller applications are simple and control-oriented, such as `photovore`. Many implement finite state machines, which are table-driven and require little RAM. Applications that may require more RAM are those based on

```
; This program was originally developed for controlling a LEGO
; MINDSTORMS robot so that it will find a source of light on the floor
; (flashlight, candle, white paper, etc).

(define narrow-sweep 20) ; width of a narrow "sweep"
(define full-sweep 70) ; width of a full "sweep"
(define light-sensor 1) ; light sensor is at position 2
(define motor1 0) ; motor 1 is at position A
(define motor2 2) ; motor 2 is at position C

(define (start-sweep sweeps limit heading turn)
  (if (> turn 0) ; start to turn right or left
      (begin (motor-stop motor1) (motor-fwd motor2))
      (begin (motor-stop motor2) (motor-fwd motor1)))
  (sweep sweeps limit heading turn (get-reading) heading))

(define (sweep sweeps limit heading turn best-r best-h)
  (write-to-lcd heading) ; show where we are going
  (if (= heading 0) (beep)) ; mark the nominal heading
  (if (= heading limit)

      (let ((new-turn (- turn))
            (new-heading (- heading best-h) ))
        (if (< sweeps 20)
            (start-sweep (+ sweeps 1)
                         (* new-turn narrow-sweep)
                         new-heading
                         new-turn)
            ; the following call is replaced by #f in the modified version
            (start-sweep 0
                         (* new-turn full-sweep)
                         new-heading
                         new-turn)))

      (let ((reading (get-reading)))
        (if (> reading best-r) ; high value means lots of light
            (sweep sweeps limit (+ heading turn) turn reading heading)
            (sweep sweeps limit (+ heading turn) turn best-r best-h)))))

(define (get-reading)
  (- (read-active-sensor light-sensor))) ; read light sensor

(start-sweep 0 full-sweep 0 1)
```

Figure 3: The source code of the `photovore` program.

multi-threading and those involved in data processing such as acquisition, retransmission, and, particularly, encoding (e.g. compressing data before transmission).

7.1 Platforms

Two platforms were used for experiments. We used a Linux workstation with a a 733 MHz Pentium III processor and `gcc` version 2.95.4 for compiling the C program generated by PICBIT. This allowed quick turnaround for determining the minimal RAM required by each program and direct comparison with BIT.

We also built a test system out of a PIC18F6720 microcontroller clocked with a 10 MHz crystal. We chose the PIC18F6720 rather than the PIC18F6520 because the larger RAM and ROM allowed experimentation with RAM sizes above 2 kilobytes and with programs requiring more than 32 kilobytes of ROM. Note that because of its smaller size the PIC18F6520 can run 4 times faster than this (i.e. at 10 MIPS with a 40 MHz clock). In the table of results we have extrapolated the time measurements to the PIC18F6520 with a 40 MHz clock (i.e. the actual time measured on our test system is 4 times larger). The ROM of these microcontrollers is of the FLASH type that can be reprogrammed several times, making experimentation easy.

C compilation for the PIC was done using the Hi-Tech PICC-18 C compiler version 8.30 [5]. This is one of the best C compilers for the PIC18 family in terms of code generation quality. Examination of the assembler code generated revealed however some important weaknesses in the context of PICBIT. Multiplying by 3, for computing the byte address of a 24 bit cell, is done by a generic out-of-line 16 bit by 16 bit multiplication routine instead of a simple sequence of additions. Moreover, big `switch` statements (such as the byte-code dispatch) are implemented with a long code sequence which requires over 100 clock cycles. Finally, the C compiler reserves 234 bytes of RAM for internal use (e.g. intermediate results, parameters, local variables) when com-

		PICBIT			BIT	
		Min	Byte-	ROM	Min	Byte-
Program	LOC	RAM	code	req.	RAM	code
`empty`	0	238	963	21819	2196	1296
`photovore`	38	294	2150	23050	3272	1552
`thread`	44	415	5443	23538	2840	1744
`all`	173	240	11248	32372	2404	5479
`earley`	653	2253	19293	35329	7244	6253
`interp`	800	1123	17502	35525	4254	7794

Table 2: Space usage in bytes for each system and program.

piling the test programs. Note that we have taken care not to use recursive functions in PICBIT's runtime, so the C compiler may avoid using a general stack. We believe that a hand-coding of the system in assembler would considerably improve performance (time and RAM/ROM space) but this would be a major undertaking due to the complexity of the virtual machine and portability would clearly suffer.

7.2 Memory Usage

Each of the programs was compiled with BIT and with PICBIT on the Linux workstation. To evaluate the compactness of the code generated, we measured the size of the byte-code (this includes the table of constants and the ROM space they occupy). We also determined what was the smallest heap that could be used to execute the program without causing a heap overflow. Although program execution speed can be increased by using a larger heap it is interesting to determine what is the absolute minimum amount of RAM required. The minimum RAM is the sum of the space taken by the heap, by the GC mark bits, by the Scheme global variables, and the space that the PICC-18 C compiler reserves for internal use (i.e. 234 bytes). The space usage is given in Table 2. For each system, one column indicates the smallest amount of RAM needed and another gives the size of the byte-code. For PICBIT, the ROM space required on the PIC when compiled with the PICC-18 C compiler is also indicated.

The RAM requirements of PICBIT are quite small. It is possible to run the smaller programs with less than 512 bytes of RAM, notably `photovore` which is a realistic application. RAM requirements for PICBIT are generally much smaller than for BIT. On `earley`, which has the largest RAM requirement on both systems, PICBIT requires less than 1/3 of the RAM required by BIT. BIT requires more RAM than is available on the PIC18F6520 even for the `empty` program.

The size of the byte-code and constants is up to 3 times larger for PICBIT than for BIT. The largest programs (`earley` and `interp`) take a little more than 32 KB of ROM, so a microcontroller with more memory than the PIC18F6520 is needed. The other programs, including `all` which includes the complete Scheme library, fit in the 32 KB of ROM available on the PIC18F6520.

Under the tight constraints on RAM that we consider here, even saving space by eliminating Scheme global variables is crucial. Indeed, large programs or programs that require the inclusion of a fair part of the standard library use many global variables. Fortunately, the optimizations performed by our byte-compiler are able to remove almost all of them. Table 3 indicates the contribution of each program transformation at eliminating global variables. The first column indicates the total number of global variables found in the user program and the library. The second one indicates how many remain after useless function elimination (UFE). The third one indicates how many remain after useless global variables have been eliminated (UGE). Clearly, considerable space would be wasted if they were kept in the executable.

Program	In sources	After UFE	After UGE
`empty`	195	0	0
`photovore`	210	43	0
`thread`	205	92	3
`all`	195	195	1
`earley`	231	142	0
`interp`	302	238	2

Table 3: Global variables left after each program transformation.

RAM size	Total run time	Avg. GC interval	Avg. GC pause time
512	84	0.010	0.002
1024	76	0.029	0.005
1536	74	0.047	0.007
2048	74	0.066	0.009
2560	74	0.085	0.011
3072	74	0.104	0.013

Table 4: Time in seconds for various operations as a function of RAM size on the `photovore` program.

7.3 Speed of Execution

Due to the virtual machine's use of dynamic memory allocation, the size of the RAM affects the overall speed of execution even for programs that don't perform explicit allocation operations. This is an important issue on a RAM constrained microcontroller such as the PIC. Garbage collections will be frequent. Moreover, PICBIT's blocking collector processes the whole heap at each collection and thereby introduces pauses in the program's execution that deteriorate the program's ability to respond to events in real-time.

We used `photovore`, a program with soft real-time requirements, to measure the speed of execution. The program was modified so that it terminates after 20 sweep iterations. A total of 2791008 byte-codes are executed. The program was run on the PIC18F6720 and an oscilloscope was used to measure the total run time, the average time between collections and the average collection pause. The measures, extrapolated to a 40 MHz PIC18F6520, are reported in Table 4.

This program has few live objects throughout its execution and all collections are evenly spaced and approximately the same duration. The total run time decreases with RAM size but the collection pauses increase in duration (because the sweep phase is proportional to the heap size). The duration of collection pauses is compatible with the soft real-time constraints of `photovore` even when the largest possible

RAM size is used. Moreover the collector consumes a reasonably small portion (12% to 20%) of the total run time, so the program has ample time to do useful work. With the larger RAM sizes the system executes over 37000 byte-codes per second.

The `earley` program was also tested to estimate the duration of collection pauses when the heap is large and nearly full of live objects. This program needs at least 2253 bytes of RAM to run. We ran the program with slightly more RAM (2560 bytes) and found that the longest collection pause is 0.063 second and the average time between collections is 0.085 second. This is acceptable for such an extreme situation. We believe this to be a strong argument that there is little need for an incremental collector in such a RAM constrained system.

To compare the execution speed with other systems we used PICBIT, BIT, and the Gambit interpreter version 3.0 on the Linux workstation to run the modified `photovore` program. PICBIT and BIT were compiled with "`-O3`" and a 3072 byte RAM was used for PICBIT, and a 128 kilobyte heap was used for BIT (note that BIT needs more than 3072 bytes to run `photovore` and PICBIT can't use more RAM than that). The Gambit interpreter used the default 512 kilobyte heap. The run time for PICBIT is 0.33 second. BIT and Gambit are respectively 3 times and 5 times faster than PICBIT. Because of its more advanced virtual machine, we expected PICBIT to be faster than BIT. After some investigation we determined that the cause was that BIT is performing an inlining of primitives that PICBIT is not doing (i.e. replacing calls to the generic "`+`" procedure in the two argument case with the byte-code for the binary addition primitive). This transformation was implemented in an ad hoc way in BIT (it relied on a special structure of the Scheme library). We envision a more robust transformation for PICBIT based on a whole-program analysis. Unfortunately it is not yet implemented. To estimate the performance gain that such an optimization would yield, and evaluate the raw speed of the virtual machines, `photovore`'s source code was modified to directly call the primitives. The run time for PICBIT dropped to 0.058 second, making it slightly faster than Gambit's interpreter (at 0.064 second) and roughly twice the speed of BIT (at 0.111 second). The speed of PICBIT's virtual machine is quite good, especially when the small heap is taken into account.

8 CONCLUSION

We have described PICBIT, a system intended to run Scheme programs on microcontrollers of the PIC family. Despite the PIC's severely constrained RAM, nontrivial Scheme programs can still be run on the larger PIC models. The RAM space usage and execution speed is surely not as good as can be obtained by programming the PIC in assembly language or C, but it is compact enough and fast enough to be a plausible alternative for some programs, especially when quick experimentation with various algorithms is needed. We think it is an interesting environment for compact soft real-time applications with low computational requirements, such as hobby robotics, and for teaching programming.

The main weaknesses of PICBIT are its low speed and high ROM usage. The use of a byte-code interpreter, the very compact style of the library, and the intricate object representation are all contributors to the low speed. This is a result of the design choices that strongly favor compactness. The use of a byte-code interpreter allows the microcontroller to run large programs that could not be handled if they were compiled to native code. The library makes extensive use of higher-order functions and code factorization in order to have a small footprint. Specialized first-order functions would be faster at the expense of compactness. The relatively high ROM space requirements are a bit of a disappointment. We believe that the runtime could be translated into more compact native code. Barring changes to the virtual machine, improvements to the C compiler or translation by hand to assembler appear to be the only ways to overcome this problem.

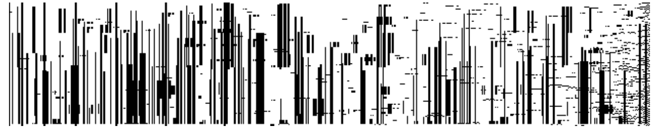

Figure 4: Heap occupancy during execution of `interp`.

PICBIT's RAM usage is the most satisfactory aspect of this work but many improvements can still be made, especially to the byte-compiler. The analyses and optimizations that it performs are relatively basic. Control-flow, type, and escape analyses could provide the necessary information for more ambitious optimizations, such as inlining of primitives, unboxing, more aggressive elimination of variables, conversion of heap allocations into static or stack allocations, stripping of useless services in the runtime, etc. The list is endless.

As an instance of future (and simple) improvement, we consider implementing a compact representation for strings and vectors intended to flatten the trees used in their representation. The representation is analogous to CDR-coding: when many consecutive cells are available, a sequence of leaves can be allocated one after the other, avoiding the need for linkage using interior nodes. The position of the objects of a sequence is obtained by pointer arithmetics relatively to a head object that is intended to indicate the presence of CDR-coding. Avoiding interior nodes both increases access speed and saves space. Figure 4 illustrates the occupancy of the heap during the execution of `interp`. The observations are taken after each garbage collection. In the graph, time grows from top to bottom. Addresses grow from left to right. A black pixel indicates the presence of a live object. There are 633 addresses in the RAM heap. The garbage collector has been triggered 122 times. One can see that the distribution of objects in the heap is very regular and does not seem to deteriorate. Clearly, there are many long sequences of free cells. This suggests that an alternative strategy for the allocation of long objects has good chances of being successful.

ACKNOWLEDGEMENTS

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada and Université

Laval.

REFERENCES

[1] Danny Dubé. BIT: A very compact Scheme system for embedded applications. In Proceedings of the Workshop on Scheme and Functional Programming (Scheme 2000), pages 35–43, September 2000.

[2] Danny Dubé, Marc Feeley, and Manuel Serrano. Un GC temps réel semi-compactant. In Actes des Journées Francophones des Langages Applicatifs 1996, January 1996.

[3] David Gudeman. Representing type information in dynamically typed language. Technical Report TR 93-27, Department of Computer Science, The University of Arizona, October 1993.

[4] Jr. Guy Steele. Data representation in PDP-10 MACLISP. MIT AI Memo 421, Massachusetts Institute of Technology, September 1977.

[5] Hi-Tech. PICC-18 C compiler (http://www.htsoft.com/products/pic18/pic18.html).

[6] Microchip. PICmicro microcontrollers (http://www.microchip.com/1010/pline/picmicro/index.htm).

[7] H. Schorr and W. Waite. An efficient machine independent procedure for garbage collection in various list structures. Communications of the ACM, 10(8):501–506, August 1967.