A Generic Virtual Machine Approach for Programming Microcontrollers: the OMicroB Project

Steven Varoumas, Benoît Vaugon, Emmanuel Chailloux

A Generic Virtual Machine Approach for Programming Microcontrollers: the OMicroB Project

Steven Varoumas[1,2], Benoît Vaugon[3] and Emmanuel Chailloux[1]

[1]Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6 UMR 7606, 4 place Jussieu 75005 Paris, France.
steven.varoumas@lip6.fr, emmanuel.chailloux@lip6.fr

[2]CÉDRIC, CNAM, 2 rue Conté F-75141 Paris Cedex 03, France.

[3]Armadillo, 46 bis, rue de la République, 92170 Vanves, France
benoit.vaugon@gmail.com

Abstract

In this paper, we present an original approach of programming microcontrollers. This approach, which stem from our first results with the OCaPIC project of running OCaml on PIC microcontrollers, consists of a generic virtual machine which goal is portability as well as memory saving. We argue that such an approach can lead to safer programs, both by using a high level programming language and by being able to use software tools dedicated to code analysis thanks to code factorization. Our generic virtual machine, called OMicroB, is foreseen to run both simple hobbyist and entertainment programs as well as critical concurrent applications in embedded systems.

Keywords: Microcontroller, Virtual Machine, Portability, OCaml, Embedded System, OMicroB, Static Analysis, Synchronous Language

1 Introduction

Microcontrollers are small programmable integrated circuits widely used in embedded systems and autonomous electronic devices. Their architecture is quite simple, and includes a set of internal memories (RAM and flash), a computing core, as well as input/output pins that allow them to interact with their environment (which can be made of sensors, actuators, other controllers, computers, ...).

Being quite affordable and energy efficient, their clock rate is modest and their memory space is scarce, compelling developers to use low level programming languages such as C or assembly language in order to carefully control the memory use of the programs they want to run on these devices. This programming approach requires developers to have a precise understanding of the microcontroller they use, and still can be difficult and error prone (with a difficult debugging process), even for a specialist. It also lacks the abstraction that higher-level programming languages like Java, OCaml or Spark/Ada can provide, such as the expressiveness of the object-oriented or functional programming paradigms, a better safety via strong typing, or even automatic memory handling.

In order to "lift" the programming of microcontrollers into a safer and more expressive programming approach, a few virtual machines (VM) have been created with the intent of running them directly on microcontrollers. These abstract machines allow developers to write programs directly in high level languages of which the bytecode is then interpreted on the chip.

In this paper, we propose OMicroB: an OCaml virtual machine written directly in the C language that we use as a kind of *portable assembly language*. OMicroB provides a set of new static analysis in order to lower memory usage, and the C language allows us to make use of various tools for code

analysis such as WCET or stack usage predictions. Our goal is to target hobbyist applications (home automation or games) as well as more critical embedded systems, on devices with limited resources.

In section 2, we present the state of the art relative to the programming of microcontrollers by the way of a virtual machine approach. In particular, we introduce OCaPIC, which is a former work intended at running the OCaml virtual machine of PIC18 devices. Section 3 introduces OMicroB and detail all of the different steps of the compilation of an OCaml source file to the uploading of a program into various microcontrollers. In section 4, we interpret some early tests with OMicroB in order to highlight the performances of the VM (in speed and memory usage), and address the subject of applications to critical software. Finally, we conclude in section 5 with a summary of the advantages of OMicroB as well as an overview of the various future works can be done on OMicroB in order to enhance its performances and expand its scope.

2 Programming Microcontrollers with High-Level Languages

2.1 Virtual Machines on Microcontrollers

Microcontrollers are devices with limited resources: their clock rate is commonly less than 100MHz and their memory space is rarely more than a few kilobytes of RAM and a hundred kilobytes of flash memory. For example, the PIC18F4620 has a clock rate of 40MHz, a RAM of 4KiB, and 64KiB of flash memory. Even worse, the ATmega328P (used in the well-known Arduino Uno boards) is limited to 2 KiB of RAM, 32KiB of flash memory, and runs at a 16MHz maximum clock rate. Nonetheless, these devices are widely used in the industry (automobile, avionics, ...) as well as in more hobbyist applications (home automation, toys, ...) due to their low cost, small size, and their efficient power management.

Their limited resources lead developers to use low level programming languages such as C or assembly language to program these devices, often needing to know precisely the architecture of the microcontroller they want to use and consequently having no freedom to change the device in the future or distribute their programs onto other systems (the instruction set between two microcontrollers of different brands is rarely inter-compatible). This way of programming, while being quite efficient is a difficult task: programming in assembly is error prone, difficult to debug, and time-consuming. Programming in C is not automatically more convenient, can lead to various runtime errors (pointer arithmetics are notoriously hard to debug), and can prevent clever code optimizations.

In order to help microcontroller programmers focus on the "logical" aspect of programming, free them of hardware considerations, and help them create safer programs, projects of developing virtual machines of higher-level programming languages for such limited microcontrollers have emerged. These virtual machines allow a safer programming approach (by taking advantage of the higher-level language features such as static typing or automatic memory management) while keeping fast and efficient programs. In fact, this approach can lead to a lighter code (including the runtime library and virtual machine) than the corresponding native code because of the more powerful and more complex instruction set of the virtual machines and thanks to bytecode compression and cleaning tools.

Among these various virtual machines, we can mention the Darjeeling Virtual Machine (DVM) [3], a port of the Java virtual machine on Atmel and ARM microcontrollers capable of running a subset of this language and featuring inheritance, threads, and garbage collection. Similarly, the PICBIT [4] and PICOBIT [9] systems are able to run Scheme virtual machines on PIC microcontrollers with powerful performances. Lastly, MicroPython [8] is a lightweight implementation of the Python 3 programming language, including a subset of its standard library. The target microcontroller of MicroPython is the STM32F405RG, which is equipped with quite substantial memory resources (1024KiB of flash ROM and 192KiB of RAM).

Some industrial solutions based on the same virtual machine approach have appeared, such as MicroEJ [7] which runs Java bytecode on platforms with a microcontroller. However, in the same way as MicroPython, these platforms offer less limited memory resources than the devices we are interested in.

2.2 OCaPIC: OCaml on PIC microcontrollers

Closer to us, the OCaPIC project [11] provides a VM capable of running OCaml bytecode in the PIC18F family of microcontrollers. The OCaPIC virtual machine is a port, written in PIC assembly language, of the original OCaml virtual machine (based on the Zinc Abstract Machine: ZAM [6]). The OCaml virtual machine is stack-based and lightweight: it only has 148 possible bytecode instructions. Besides providing classical arithmetics and control-flow instructions, it is also able to handle functional values with instructions dedicated to creating closures (pointers of code together with an environment) and applying them.

OCaPIC allows the programming of microcontrollers to take advantage of all the functionalities of the OCaml language (from version 3.12 to 4.05.0), such as its rich expressiveness (OCaml features various programming paradigms: functional, modular, imperative, and object-oriented), its automatic memory management, and its strong static typing that guarantees the absence of typing errors at runtime. OCaPIC thus offers a powerful, portable developing process, while keeping very satisfactory performances. OCaPIC comes with various tools, in particular the `ocamlclean` tool that transforms the OCaml bytecode to remove unused closures from the heap. This process of bytecode cleaning is essential for running non-trivial programs on devices with such limited memory space.

OCaPIC, as well as others VM approaches intends to increase code portability: the same bytecode is able to run in every VM developed for the language. However, it may seem quite contradictory that these VM have all been developed for a specific set of microcontrollers (sometimes in assembly language, as it is the case with OCaPIC to maximize performances), restricting the portability of the VM approach to the constraint of developing a VM for each and every device one would want to run bytecode on.

3 OMicroB: A Generic Implementation of the OCaml Virtual Machine

By benefiting from the portability of the bytecode of a higher-level language, OCaPIC has been a first step towards a generic approach to the OCaml programming of microcontrollers, but is still limited to the PIC18F family of microcontrollers. In order to run the OCaml VM on other devices (such as Atmel AVR microcontrollers which are used in Arduino boards), one would have to re-write another version of the OCaml VM for each of these devices, which can be time-consuming and feel contradictory to our goal of code portability. We thus intend to provide an even more generic approach by proposing OMicroB: a VM directly written in C language, that can be configured in order to adapt to the resources of the hosting microcontroller.

The workflow from an OCaml source file to a microcontroller executable file (represented in figure 1) is the following: first, the source file is compiled ①, and the resulting bytecode is cleaned ②. Then, the cleaned bytecode is converted into a C file ③. This C file is finally linked with our virtual machine ④ and used by a C compiler made for the right device ⑤. The resulting executable can now be transferred to the microcontroller and run. We describe these different steps in details in the following subsections.

3.1 Bytecode compiling (1) and cleaning (2)

The OCaml source file of our microcontroller's applications are in standard OCaml syntax, and thus can be used with the native OCaml bytecode compiler (`ocamlc`). The resulting file is an OCaml bytecode file that could (provided all of the external C functions that are needed by the program are implemented) still be executed on a standard PC.

This file can contain dead code (primarily unused closures, created at runtime, that come when opening an OCaml library) that could take a lot of memory and result in unusable programs because memory is a very scarce resource on microcontrollers. To eliminate this unused code, we use the

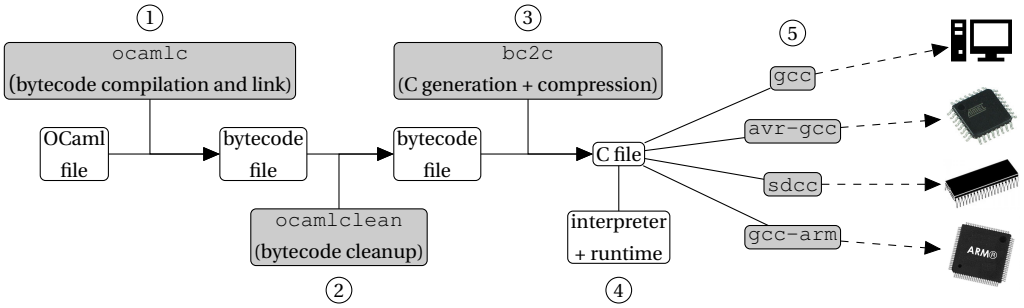


Figure 1: From an OCaml source file to a microcontroller

`ocamlclean` tool (bundled with OCaPIC [11]) that removes blocks of unused code by performing static analysis of the functional program in order to detect which closures may not be used at runtime. The resulting program uses way less memory and it is now realistic to run it into a device with limited resources.

3.2 The bytecode-to-C tool (`bc2c`) (3)

The `bc2c` tool takes as input the bytecode file output by the OCaml linker `ocamlc` or the bytecode cleaner `ocamlclean`. It then performs some code analysis and transformations to optimize and compact the bytecode, and finally produces a C source file defining the constants and static arrays needed by the OMicroB implementation of the OCaml Virtual Machine described farther.

Bytecode compaction

The bytecode "compaction", performed by `bc2c`, simply consists in a compact encoding of opcodes and arguments of the bytecode instructions. Since we target 8 bits architectures in the first place and since the bytecode is only stored in flash memory and never loaded in RAM, their is no interest in code alignment and each byte avoided in the code implies a significant gain in bytecode reading and interpretation.

Obviously, our encoding of the bytecode uses only one byte per opcode. For bytecode instructions having arguments, we adapt the number of bytes used to store them. For a single instruction like `GETGLOBAL <ind>`, we may generate different opcodes depending on the encoding of arguments. To reduce code pointer sizes stored in the bytecode, each code pointer is implemented by a relative offset instead of an absolute address. For example, in most cases, a `BRANCHIF <ofs>` instruction only takes two bytes in the bytecode, one for the opcode (corresponding to a `BRANCHIF` with an argument stored on one byte), and one for the offset as argument.

Virtual machine cleaning

Most of compiled programs do not use the whole set of bytecode instructions, in particular when some features of the language are not used like exceptions or objects, for example. Furthermore, since the implementation of instructions with arguments are replicated for the different encoding of arguments, lots of instructions are nearly never generated by `bc2c` like, for example, a `CLOSURE <ofs>` with an offset encoded on four bytes.

The set of opcodes generated by `bc2c` is then adapted to the program and the C code of the Virtual Machine is cleaned from all its unused instructions and primitives thanks to constants generated by `bc2c`. A small program is then compiled into a small bytecode and a small Virtual Machine. Moreover, to improve the speed of bytecode interpretation, the set of used opcodes is chosen to always be contiguous, from 0 to <n>-1 where <n> is then number of used opcodes.

Initialization shortcut

An OCaml program starts by the deserialization of OCaml allocated constants like strings or constant lists. It continues with the interpretation of the beginning of the bytecode, typically generated from libraries, and which its interpretation generates lots of closure allocations, module creations, global variables computations (like a constant matrix), etc. The main program then starts with its hardware initializations, inputs/outputs, etc.

The initialization of constants and libraries may be time and memory consuming, typically stack consuming when some modules exports lots of functions. Furthermore, lots of libraries are usually used partially, typically when they expose conditional codes or code generated by functor instanciations.

To reduce flash and RAM usage and make initialization faster, the `bc2c` tool simulates, on a computer at compile time, the execution of the beginning of the program until the first input/output. It allocates a stack and a heap miming the OMicroB representation of OCaml values, deserialize global constants and starts bytecode interpretation. At the first input/output, it dumps in the generated C file the living part of the heap, stack, and bytecode, i.e. the part of the bytecode accessible from the current code pointer and closures from the cleaned heap.

3.3 The bytecode interpreter and runtime (4)

The OCaml virtual machine

The OCaml virtual machine is based on a stack: this stack (with a companion accumulator - `acc` - register) stores OCaml values that are manipulated by the interpreter. In our implementation, these values can be:

- Direct values (integers, or floats).
- Block values, used for representing "boxed" values (such as tuples, lists, ...). These blocks are allocated on the program's heap.
- Pointers to a block of code in flash memory.

The virtual machine also contains several registers: the main ones being the program counter (`pc`), the stack pointer (`sp`), the accumulator (`acc`, used to avoid too much stack pop and push), a pointer to the environment (`env`) and a pointer to the global data (`global`).

Given the parametric polymorphism of the OCaml language, the different values inside the virtual machine are uniformly represented. In OMicroB, values can be configured to be stored on 32 bits or 64 bits words (independently of the actual memory architecture of the microcontroller).

Representation of the OCaml values in OMicroB

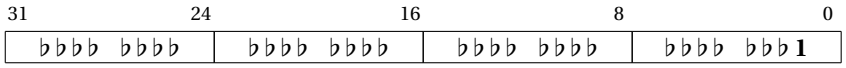
In order to be correctly handled by the interpreter, the OCaml values need be quickly distinguishable depending on their nature: integers, floating-point numbers and pointers all need a different memory representation for the program to run correctly.

Our value representation differs little from the original representation by the native (used on personal computers) bytecode interpreter (`ocamlrun`) with two main exceptions: floating-points numbers are not allocated on the heap (as it is done in the PC implementation of the runtime) and are instead direct[1]. The other difference is that one needs to discriminate between heap pointers (with the values stored in RAM) from code pointers (the values stored in the flash memory of the microcontroller).

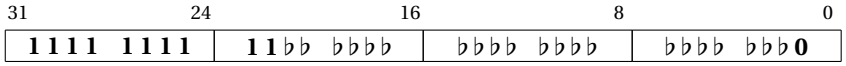
[1] This choice stems from our desire to use values of simple types (including floats) in a synchronous extension of OCaml without dealing with interruptions from the garbage collector during a synchronous instant.

Our representation of the OCaml values uses NaN boxing [5]: an encoding of values inside the space of the NaN values defined in the IEEE 754 floating-point standard. This representation is the following (on a 32 bits configuration):

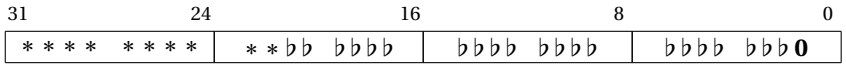
- Integers are represented on 31 bits, with the lsb[2] of the word set to 1:



- Pointers to a block on the heap have their lsb set to 0 and their first ten msb[3] set to 1.

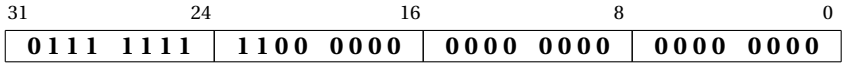


- Pointers to flash memory are represented without modification, but their adresses are limited to $2^{32}-2^{22}-1$ in order not to have only ones in the ten msb (and be mistaken for heap pointers).



(the "$*$"s cannot all be **1**)

- Floats are represented "as is" (32 bits) with the following (single) *NaN* value:



This representation is simply extended for a 64 bits configuration.

Interpreting the bytecode file

The core of OMicroB is the OCaml bytecode interpreter: each of the 148 OCaml bytecode instructions is handled by a corresponding C code that operates on the OCaml stack and registers.

For example, the chunk of bytecode on figure 2 creates a closure (that takes an integer parameter and computes another closure), applies it to the integer value 4, and applies its functional result to the integer value 8.

The interpreter reads the array representing the bytecode program (created by `bc2c`) in the C file. This array is stored in flash memory (since it is read only) in order to gain RAM space. To lower flash memory usage of the overall program, we take advantage of the C preprocessor by defining macros for each OCaml instruction and only compile the code of the instructions that are used in the given bytecode. That way, the interpreter is adapted to the program it must handle, and doesn't come with unneeded code.

The runtime library

In order to run non-trivial programs, OMicroB comes with a runtime library that defines various predefined types such as lists, arrays, references (mutable variables) or strings, and a generic compare function. Just like the original OCaml virtual machine, OMicroB can also be extended with external C functions that can be useful when porting the VM to other microcontroller architectures (typically, the input/output primitives of a given microcontroller will be written in C and use in the OCaml code just as any OCaml function).

Memory management of the heap is provided by a very simple garbage collector (GC) that implements a *stop-and-copy* algorithm. This GC is tail recursive (except for the handling of mutually recursive value), doesn't allocate any memory for itself, and is very quick, but it has the important drawback of only allowing the use of half the heap space for all heap allocations.

[2] least significant bit
[3] most significant bit

```
let add_x x =                 (* add_x takes an integer x and              *)
  (function y -> y + x)       (* creates a functional value that takes y   *)
in                            (* and computes y + x                        *)
  let add_4 = add_x 4 in      (* add_4 is the function: (function y -> y + 4) *)
  add_4 8                     (* = 12                                      *)
```

```
56 BRANCH 62
```
go to address 62

```
57 GRAB 1
```
create a closure (which code begins at address 58) into the stack and go back to the caller (i.e. return at address 66)

```
58 ACC 0
```
put the top value of the stack into `acc` (the accumulator)

```
59 PUSHACC 2
```
push the content of `acc` into the stack and put the third value from the top of the stack into `acc`

```
60 ADDINT
```
add the first and second value of the stack and put the result in `acc`

```
61 RETURN 2
```
end of the closure

```
62 CLOSURE 0 57
```
create a closure (which code begins at address 57) into `acc`

```
63 PUSHCONST 4
```
push the content of acc into the stack and put the value 4 into `acc`

```
64 PUSHACC 1
```
push the content of `acc` into the stack and put the second value from the top of the stack into `acc`

```
65 APPLY 1
```
go to the code of the closure that is inside `acc` (i.e. the code at address 57) and put the result (corresponding to `function y -> y + 4`) into `acc`

```
66 PUSHCONST 8
```
push `acc` and put 8 in `acc`

```
67 PUSHACC 1
```
push `acc` and put the second value from the top of the stack into `acc`

```
68 APPLY 1
```
go to the code of the closure that is inside `acc`, and put the result into `acc`

Figure 2: Example of an OCaml file and its corresponding bytecode

3.4 Compiling the C program (5)

Compiling the resulting C program is straightforward: we use a compiler adapted to the desired microcontroller (e.g. `avr-gcc` for Atmel microcontrollers, `sdcc` for PIC, ...). The main part of the C program that must be different between families of devices is located in the runtime: the needed C primitives called by the virtual machine (for example those needed to switch the values on the microcontroller's pins) often need to be written differently depending on the used architecture.

4 Performances and application to critical software

We conducted different tests of OCaml programs using our OMicroB virtual machine on the Arduino Uno board, a common hobbyist device holding an AVR Atmega328p microcontroller with a 16 MHz clock rate, 2 KiB or RAM, and a flash memory of 32 KiB.

The different executed tests are the following:

- `empty` is the empty program, doing nothing. Note that since this program does nothing, and thanks to bc2c's instructions cleaning, the generated source doesn't contain any code responsible for handling any OCaml instruction, and the executable is thus small (it contains only the code of the runtime). Disabling instruction cleaning, the VM total footstep (interpreter + GC + runtime) would be around 22 KiB, so the gain brought by the bc2c analysis is substantial.

- `oddeven` checks if the integers between 0 and 100 are odd or even numbers using two mutually recursive functions. This programs is intended to test (mutual) recursion.

- `sieve` computes 100 times the prime numbers inferior to 10 using the sieve of Eratosthenes method. This program tests the pattern matching mechanism as well as the GC of the runtime.

– `deriv` computes 100 times the symbolic derivative of x^2 over x. This is intended to test more complex pattern matching, the GC, and the representation of strings.

– `integr` computes 100 times the integral $\int_0^1 x^2 \, dx$. This program tests the float implementation and the GC.

The table of figure 3 displays the various space and time results of the execution of these programs: the sizes of the OCaml bytecodes, the sizes of their executable files, the speed of their execution on the Uno, the number of instructions interpreted by the VM during execution, the deduced number of instructions per second, the initial RAM use of the program, and the configuration of OMicroB for the stack size and heap size.

Program	**Bytecode size (B)**	**Exec. size (B)**	**Time (ms)**	**# of insts**	**Speed (inst/s)**	**Init. RAM usage (B)**	**Stack size (words)**	**Heap size (words)**
`empty`	0	4270	0	0	_	64	0	0
`oddeven`	66	5318	195	31 834	163 251	1003	64	64
`sieve`	189	9646	476	53 247	111 863	1825	64	164
`deriv`	261	8846	71	7391	104 098	1946	28	196
`integr`	132	9722	212	21 936	103 471	1849	64	164

Figure 3: Execution time and space of various OCaml programs

These different results validate our generic approach: the generated code (containing the virtual machine and its runtime, as well as the bytecode of the program) can fit in the very limited memory space offered by the Arduino Uno. Our automatic dynamic memory management is able to efficiently reuse memory: for example, the `integr` program allocates 100 closures of 5 words in total (i.e. 16 KiB) during its execution, but the program only needs one closure at every moment: the GC cleans the heap accordingly 3 times during execution. The `sieve` and `deriv` programs perform respectively 50 and 16 garbage collection passes during their execution.

The cost of this portable approach is difficult to precisely evaluate, as comparing the previous results with OCaPIC might not be particularly relevant: hardware differences between the architectures of the Atmel AVR of the Arduino and the PIC of OCaPIC makes comparison unsuitable, as for example reading a byte in flash memory takes 12 cycles on AVR while it takes only 2 on a PIC. Moreover, we handle in OMicroB values stored on 32 bits words, whereas OCaPIC uses 16 bits values, and the small size of the Uno's flash memory implies frequent triggers of the GC, thus a decrease in speed is expected: OMicroB on AVR is about 3.7 times slower than OCaPIC on a PIC18F4620 (due in parts to these differences in value representation, cost of frequent flash memory access, and heap space).

We experimented with porting OMicroB to PIC18 microcontrollers, unfortunately the performances of the various C compilers for 8 bits PIC microcontrollers (`sdcc` or `xc8`) were quite poor and their optimizations not very satisfactory (e.g. the `switch` instruction that discriminates between the different bytecode instructions was compiled by `xc8` into several nested `if` instructions): our speed results fell at least 5 times slower than the heavily optimized VM of OCaPIC when using these compilers.

The generic approach of using a virtual machine written in C comes with the price of predictable but acceptable decrease in speed (which can be - as we have noticed with PIC - greatly dependent of the C compiler), and we could certainly improve the speed performances of the VM by writing some target-specific code for the parts that we observed as badly optimized by compilers, as well as implementing a more efficient garbage collection algorithm. However, speed was not our primary goal, as our target market are hobbyists applications (that will benefit from the programming level and the ease of debugging applications), and industry embedded programs which will take advantage of the safety offered by the OCaml language, via its static typing and analysis tools. The targeted applications won't need high speed performances - since native programs are hardly beatable in speed anyways - and we instead focused our efforts in reducing memory use (the targeted architectures might be 100 times slower than PCs, but their memory resources are 1 000 000 times smaller). Small programs might

be slower than their equivalent in C, but thanks to the terseness of bytecode instructions, the total size of the entire program (VM + runtime + bytecode) will be smaller for more complex programs (and thus fit in very limited space).

Application to critical systems

In [10] we argued that a synchronous programming model was very well suited for concurrent programming of microcontrollers, especially for critical applications. A synchronous model assures us that no new values need to be allocated on the heap during execution, that way the garbage collector cannot be triggered during execution of a synchronous instant and this is a major advantage for Worst-Case Execution Time (WCET) and memory use analysis for critical programs (as well as speed performances).

For example, the following synchronous function (or *node*) written in OCaLustre (a data-flow synchronous extension of OCaml) defining a counter can easily be handled by OMicroB (for clarity, we give on the right side the translation of this function into the Lustre synchronous language):

```
let%node count (init) ~return:(c) =
  c = init ->> (c + 1)
```

```
node count (init:int) returns (c:int);
let
        c = init fby c + 1
tel
```

A program executing this node for 1000 loops has the following memory and speed results (note that the heap use of such a program is very light - only 10 words are needed):

Bytecode size (B)	Executable size (B)	Time (ms)	# of insts	Speed (inst/s)	Init. RAM usage (B)	Stack size (words)	Heap size (words)
77	6546	153	25 030	163 594	389	16	10

The use of both OCaml for programs and C for the VM offers a way to factorize code analysis by running different tools on each levels: the bytecode can be debugged using the traditional OCaml debugger (`ocamldebug`) or can be profiled without having to flash the microcontroller by compiling and running the VM on a Unix machine using a native C compiler (and a library simulating the I/O of the chip). Using C tools, execution time can be bounded with the help of a WCET analyser such as Bound-T [2] or the OTAWA framework [1] by computing an execution time for each bytecode instruction. Stack usage of the VM can also be computed using C or assembly tools. We were successful in computing WCET of each OCaml instructions for an Atmel target with the help of Bound-T.

5 Conclusion and future work

We described a virtual machine approach of programming microcontrollers, using an implementation of the OCaml virtual machine with various tools dedicated to code size and memory use compaction. We were able to run the entire VM on very limited devices, with good performances. The same performances are expected for bigger programs running on similar devices, but with more memory space.

Our generic virtual machine approach guarantees portability: we were successful in running simple programs on Atmel AVR microcontrollers, experiment with PIC microcontrollers, and debug on PC. This approach offers access to various levels of factorization: bytecode analysis can be done without information about the target device, and C tools can be used to analyze compiled programs.

We argued in [10] that programs for microcontrollers are inherently concurrent. We proposed OCaLustre, a lightweight synchronous extension to OCaml. This extension is even more promising using this generic VM approach because this factorization of code can be taken advantage of to compute the WCET of each bytecode instruction appearing during a synchronous instant.

OMicroB is dedicated to target simple non critical programs (such as games) in very limited hardware, as well as critical embedded applications, for example in the LCHIP[4] project (a low cost safe execution platform based on redundant PIC32 microcontrollers, where our VM approach would bring a

[4] http://www.clearsy.com/2016/10/4260/

diversified execution for redundancy). Those two targets can overlap on safety needs, as it was shown with the specification (and its check) of a Tetris game for the Arduboy board in SPARK[5].

The results of our experiments with OMicroB are very promising, and we intend to improve its performance and target more devices: we are interested in experimenting with the PIC32 microcontrollers used in the LCHIP project (their compilers based on GCC might offer better code optimizations than on PIC8). To offer even better performances, the implementation of a Mark & Compact garbage collector that wouldn't "waste" half of the heap is a work in progress. In `bc2c`, static analysis by abstract interpretation could be performed so that we could detect global data that are never modified (such as strings and closures) and move them at compile-time from the RAM of the microcontroller to its (typically larger) flash memory. With this optimization, instead of dumping one heap, `bc2c` should generate three heaps: an immutable heap stored in the flash memory containing constant and unmovable blocks known at compile time, a mutable heap in the RAM memory containing mutable but unmovable blocks known at compile time and pointed by the initial stack and the immutable heap, and the standard heap in RAM, initially empty, used to store blocks allocated at runtime and managed by the garbage collector. These different improvements will allow to run more complex programs on devices with a small memory space and will also be useful on more powerful microcontrollers.

References

[1] BALLABRIGA, C., CASSÉ, H., ROCHANGE, C., AND SAINRAT, P. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *Workshop on Software Technologies for Embedded and Ubiquitous Systems, SEUS 2010* (Oct. 2010), pp. 35–46.

[2] BOUND-T. Bound-T time and stack analyser - `http://www.bound-t.com`.

[3] BROUWERS, N., CORKE, P., AND LANGENDOEN, K. Darjeeling, a java compatible virtual machine for microcontrollers. In *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion* (New York, NY, USA, 2008), Companion '08, ACM, pp. 18–23.

[4] FEELEY, M., AND DUBÉ, D. Picbit: a Scheme system for the PIC microcontroller. In *Scheme and Functional Programming Workshop (SFPW'03)* (Nov. 2003), pp. 7–15.

[5] GUDEMAN, D. Representing type information in dynamically typed languages. Tech. Rep. TR 93-27, Dept of Computer Science, The University of Arizona, Oct. 1993.

[6] LEROY, X. The ZINC experiment : an economical implementation of the ML language. Tech. Rep. RT-0117, INRIA, Feb. 1990.

[7] MICROEJ. Embedded Software Solutions for IoT Devices - `http://www.microej.com`.

[8] MICROPYTHON. Python for microcontrollers - `https://micropython.org`.

[9] ST-AMOUR, V., AND FEELEY, M. PICOBIT: A Compact Scheme System For Microcontrollers. In *Proceedings of the 21st international conference on Implementation and application of functional languages* (2010), IFL'09, Springer-Verlag, pp. 1–17.

[10] VAROUMAS, S., VAUGON, B., AND CHAILLOUX, E. Concurrent Programming of Microcontrollers, a Virtual Machine Approach. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)* (2016).

[11] VAUGON, B., WANG, P., AND CHAILLOUX, E. Programming Microcontrollers in OCaml: the OCaPIC Project. In *International Symposium on Practical Aspects of Declarative Languages (PADL 2015)* (June 2015), no. 9131 in LNCS, Springer Verlag, pp. 132–148.

[5] http://blog.adacore.com/spark-tetris-on-the-arduboy