Towards Self-Hosting Software for Small Computers

Lionel Morel Guillaume Salagnac Nicolas Stouls
firstname.lastname@insa-lyon.fr
INSA Lyon, Inria, CITI, UR3720, 69621 Villeurbanne, France

Abstract

In this paper, we show by example that software development can be carried out on much smaller platforms than is usually deemed necessary, e.g. a PC. First, we identify requirements for a self-sufficient software system, namely programmability, self-hosting, and portability. Then, we study the design and implementation of CollapseOS, a bare-metal Forth system written by Virgil Dupras and show that it satisfies these requirements. Finally, we evaluate its footprint in terms of computing resources: CollapseOS needs about 6kB of flash memory for machine code, about 180kB of storage for source code, and it needs approximately 12kB of RAM to fully recompile itself.

1 The Need for Small Computers

The quest for endless growth is a key feature of modern societies in general, and of the ICT industry in particular. But this trajectory is not sustainable. For decades, computing hardware has been improving exponentially: faster processors, more memory capacity, better network bandwidth, etc. Recently however, this trend has been slowing down, as we approach fundamental physical limits [5]. Even the current state of technology might not last forever because we gradually deplete the planet's reserve of mineral resources. Also, the environmental impact of ICT has reached significant levels [7] and we should not let it get any worse. In contrast, most climate scientists advocate [9] for degrowth, in the hope of bringing human activities under our planet's sustainability limits.

With this in mind, one may ask whether the benefits brought by digital technology are worth keeping the industry going at all. In this work, we make the assumptions that it is indeed feasible to produce computers in a sustainable way, and that these systems would be much less powerful than today. From there, we explore the issues involved in producing software for such machines. This looks similar to the challenges of embedded computing, i.e. computers integrated in so-called into "smart objects". But current practice is to perform all software engineering on an ordinary machine, and transfer only the resulting executable to the target platform.

In this paper, on the other hand, we study what it means and what it costs to produce software if there are only small computers available. First, we identify three properties that are necessary for a software system to be considered self-sufficient. Programmability, i.e. what tools really are necessary to write and run programs ? Self-hosting, i.e. can we use this toolset to maintain and further develop the tools themselves? Portability, i.e. can the same source code target different hardware platforms? Second, we study the design and implementation of CollapseOS [6] (abbreviated COS from here on) a bare-metal Forth system developed by Virgil Dupras for 16-bit machines. We show how it meets our three self-sufficiency requirements. Finally, we describe how we ported the OS to a new platform, and we evaluate its footprint in terms of computing resources.

2 Operating Systems for Small Computers

In this paper, we say that an operating system (OS) is *programmable* when it lets users write and run new programs from within the platform itself. For instance, GNU/Linux distributions usually include all tools required for software development: text editor, compiler, debugger, etc. However, not all OSes are programmable in this sense: Embedded Operating Systems like FreeRTOS [8] only provide a runtime platform for embedded programs, but all programming tools run on a Linux machine. To be more specific, a programmable OS must offer ways to: *write* programs as source code; *transform* them into some executable form; and let users *execute* and *debug* these programs how and when they please.

To be relevant for this study, an OS must also be *self-hosted*: the programming tools are available on the platform, and the system is able to fully recreate a new executable version of itself, without depending on another machine. While this is the case for most Linux systems, Android based systems don't offer this possibility.

In a world where no computer is powerful enough to run Linux, all OSes will probably have to be self-hosted. Moreover, we claim that *source code portability* also remains an essential property, because writing specific programs for each platform would require too much effort to be practical. There are many small, self-hosted OSes available today but they are typically not portable. For instance, projects like SectorForth [3] and SectorLisp [11] are bare-metal language interpreters distributed as tiny executable files, typically 512 bytes of x86 machine code each. Their main originality is meta-circularity i.e. the interpreter comes with its own source code, and as such is able to re-generate its own executable. But both projects are tied to the x86 architecture: the source code is actually x86 assembly with an ad-hoc syntax. Also, they rely heavily on BIOS/UEFI services, so their actual bare-metal nature is questionable.

Many programming languages have small and/or bare-metal and/or portable implementations [12] but they typically have, in one form or the other, some dependency on other, much larger, software systems. In a hypothetical world with only small computers, we claim that a programmable, self-hosted, portable OS must be available.

3 Technical Background: Forth and CollapseOS

Forth was created in the 1970s to write portable bare-metal control software for scientific equipment [4]. A Forth program is a sequence of whitespace-separated *words*, to be interpreted in Polish postfix notation, e.g. "`6 4 3 + *`" produces 42 on the *operand stack*. Absence of syntax makes for efficient implementation: words are simply *executed* one after the other. To do this, the interpreter looks up each word in *the dictionary*, a global data structure that associates names with their implementations. The dictionary starts pre-populated with *core words* which implement arithmetic e.g. "`+`", basic stack operations e.g. "`DUP`", memory accesses, etc. Of course, the user can also write new *definitions*. The main user interface

to a Forth system is an interactive read-eval-print loop (REPL). Most systems also offer persistent storage in the form of a *Block File System* (BFS), a flat array of 1024-bytes blocks. On bare-metal, blocks will typically be mapped directly onto the underlying medium.

CollapseOS [6] is a small Forth OS developed by Virgil Dupras since 2021. Its implementation is minimalistic: the dictionary is a simply linked list, and core words are implemented as raw machine code. Executing a word consists in only two simple steps: search for its name in the list, then have the CPU jump to its body. Every word body must end with a jump back to the interpreter loop. But not all words are core words. The body of a *compiled word* consists not in machine code but in a list of pointers to other words. The interpreter executes such words as direct-threaded code [2]: every pointer is invoked as a *subroutine*, so that control can return back afterwards. All these mechanisms are fully exposed to the programmer, who can freely manipulate the internal state of the interpreter. This makes it possible to implement higher level constructs like `IF...THEN`.

The main strength of COS is that everything is implemented in Forth, from machine code generation, to word compilation, name lookup and the REPL itself. Core words, assemblers, and device drivers are provided for several platforms: Z80, 8086, 6502, 6809, and AVR.

4 Contribution: A Study of CollapseOS

We now discuss how COS satisfies all the properties described in section 2. To gain a solid understanding of the issues of portability and self-hosting, we carried out a complete port of the code to a new architecture (MSP430).

COS offers crude but self-sufficient *programming tools*, all written in Forth. There are two text editors: `ED` is a line editor and `VE` works on a whole BFS block. The system provides facilities to `LOAD` and `COMPILE` source code from the BFS. Programs are launched from the REPL prompt. The user can thus read and modify the whole source code from within a running system. COS however offers very limited debugging capabilities: beyond rudimentary stack overflow/underflow checks, the system (including the compiler) has no error checking whatsoever. As a result, even simple mistakes like forgetting the final semicolon in a definition result in crashes or weird behaviors, mostly abrupt reboots. For comfort, we thus relied heavily on traditional unix tools: emacs, bash, make, etc. We validated everything against real hardware, but in practice we spent a lot of time within the `mspdebug` [1] MSP430 emulator to take advantage of step-by-step execution, memory inspection, etc.

Self-hosting, one of the main COS design goals, is achieved by having almost no abstractions in the implementation. For instance, a dictionary entry is stored in memory as just: its name in ASCII, one byte for name length, a linked list pointer, and then bare machine code. That way, Forth programs interact at low level with the dictionary, without any runtime support. The compiler is literally five lines of code, of which four are a while-loop: read a word name, look it up in the list, write the pointer to memory, and repeat. Similarly, the assembler is just a set of routines generating machine code in binary. The interpreter engine is implemented in terms of this assembly language. Core words are written as a series of Forth routines which, when executed, each generate a new dictionary entry. By supplying the right BFS blocks to the compiler in the right order, one can build a complete new binary image (interpreter+dictionary) in memory. This executable can then be stored e.g. in flash, so that rebooting will pick up on the update. Loading the compiler, and feeding it blocks, is itself all programmed in Forth.

Portability comes for free, as all the facilities described previously work just as well for a different target architecture. One just needs to load the proper assembler, then use it to generate the interpreter engine and core words. Porting COS simply consists in writing these platform-dependent parts. Our *assembler* implements about 50 MSP430 instructions in 70 lines of Forth. The *interpreter engine* is less than 10 lines of assembly. The 35 *core words* (`+`, `DUP`, etc) represent about 200 lines of assembly. As for the *drivers*, we implemented the bare minimum: reading and writing flash memory blocks, and a serial port for user input, for a total of about 100 lines of assembly. The rest of COS (about 180 compiled words) consists in about 350 lines of platform-independent Forth.

We now evaluate the *computing resources* required by COS to recompile (or cross-compile) itself. The executable image is around 6kB, including the interpreter engine, core words, etc. On a microcontroller like the MSP430, code is stored in flash and executed in-place by the CPU. We implemented the BFS in a separate region of flash memory. COS's platform-independent source code is about 2kloc, or 5kloc total if we add code for all supported platforms. This represents less than 180kB of data, even though in practice it is spread into about 300 BFS blocks of 1kB each. In terms of RAM, the high watermark during self-recompilation is just under 12kB, decomposed as follows. Newly loaded code (cross-compiler, assembler, etc) occupies about 4.5kB of dictionary space. The runtime state of the system itself is about 1.5kB: execution stacks, global variables, BFS buffer. The newly created image itself occupies about 6kB. The whole process takes about $2{\times}10^9$ MSP430 CPU cycles and less than a minute of simulation time. With the default 1MHz clock speed of a real MSP430, this would amount to approximately 35 minutes.

5 Conclusion

In this paper, we describe requirements for building self-sufficient software systems: programmability, self-hosting and portability. With these properties, it becomes possible to program small machines without depending on bigger ones. More specifically, we show that a self-sufficient programming environment can fit on a machine with as little as 12kB of RAM. We note that such a small size brings additional benefits: the entire code base fits in a single human brain, empowering users to adapt the system to their needs, to port it to other platforms, etc.

The perspectives of this work are numerous. Obviously, we still have to answer the question of whether digital devices can be produced sustainably at all [10]. If yes, then we will investigate what the most powerful sustainable hardware would be. If that computer happens to be smaller than our target, then the current paper is irrelevant and the question becomes "can we still program within less than 12kB?". On the other hand, if that "sustainability limit" turns out to be higher, then a variety of interesting questions open up for research. For instance, many features of modern languages are lacking from Forth, e.g. typing, error handling, concurrency, debugging. We want to explore the cost/benefit ratio of these features.

References

[1] Daniel Beer. 2017. MSPDebug, a free debugger for use with MSP430 MCUs. https://dlbeer.co.nz/mspdebug/. (2017).

[2] James R. Bell. 1973. Threaded code. *Communications of the ACM*, 16, 6, (June 1973), 370–372.

[3] Cesar Blum. 2020. SectorForth, a 16-bit x86 Forth that fits in a 512-byte boot sector. https://github.com/cesarblum/sectorforth. (2020).

[4] Leo Brodie. 2004. *Thinking forth*. Punchy Publishing.

[5] Ralph K Cavin, Paolo Lugli, and Victor V Zhirnov. 2012. Science and engineering beyond moore's law. *Proceedings of the IEEE*, 100, Special Centennial Issue, 1720–1749.

[6] Virgil Dupras. 2021. Collapse OS, Bootstrap post-collapse technology. http://collapseos.org/. Accessed: 2024-09-12. (2021).

[7] Charlotte Freitag, Mike Berners-Lee, Kelly Widdicks, Bran Knowles, Gordon S Blair, and Adrian Friday. 2021. The real climate and transformative impact of ICT: A critique of estimates, trends, and regulations. *Patterns*, 2, 9.

[8] Fei Guan, Long Peng, Luc Perneel, and Martin Timmerman. 2016. Open source FreeRTOS as a case study in real-time operating system evolution. *Journal of Systems and Software*, 118, 19–35.

[9] Intergovernmental Panel on Climate Change (IPCC). 2022. *Climate Change 2022: Mitigation of Climate Change. Contribution of Working Group III to the Sixth Assessment Report of the Intergovernmental Panel on Climate Change, Technical Summary*. Cambridge University Press.

[10] Nicolas Moreau, Thibault Pirson, Grégoire Le Brun, Thibault Delhaye, Georgiana Sandu, Antoine Paris, David Bol, and Jean-Pierre Raskin. 2021. Could Unsustainable Electronics Support Sustainability? *Sustainability*, 13, 12.

[11] Justine Tunney. 2021. SectorLisp, a 512-byte implementation of LISP that's able to bootstrap John McCarthy's meta-circular evaluator on bare metal. https://github.com/jart/sectorlisp. (2021).

[12] Samuel Yvon and Marc Feeley. 2021. A small scheme VM, compiler, and REPL in 4k. In *Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (VMIL 2021). Association for Computing Machinery, Chicago, IL, USA, 14–24. ISBN: 9781450391092. DOI: 10.1145/3486606.3486783.