Self-Contained Development Environments

Or Free Yourself from the Yoke of Operating Systems

Guido Chari
Czech Technical University

Javier Pimás
Palantir Solutions S.R.L.

Olivier Flückiger
Northeastern University

Jan Vitek
Northeastern University & CTU

Abstract

Operating systems are traditionally implemented in low-level, performance-oriented programming languages. These languages typically rely on minimal runtime support and provide unfettered access to the underlying hardware. Tradition has benefits: developers control the resources that the operating system manages and few performance bottlenecks cannot be overcome with clever feats of programming. On the other hand, this makes operating systems harder to understand and maintain. Furthermore, those languages have few built-in barriers against bugs. This paper is an experiment in side-stepping operating systems, and pushing functionality into the runtime of high-level programming languages. The question we try to answer is how much support is needed to run an application written in, say, Smalltalk or Python on bare metal, that is, with no underlying operating system. We present a framework named NopSys that allows this, and we validate it with the implementation of CogNos a Smalltalk virtual machine running on bare x86 hardware. Experimental results suggest that this approach is promising.

ACM Reference Format:
Guido Chari, Javier Pimás, Olivier Flückiger, and Jan Vitek. 2018. Self-Contained Development Environments: Or Free Yourself from the Yoke of Operating Systems. In *Proceedings of Submitted*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Operating systems are software layers shielding applications from direct contact with the underlying hardware while, at the same time, acting as resource managers. Traditionally they consist of large and complex code bases written in performance-oriented languages such as C. The choice of language is not accidental. As the operating system is the bottom layer of the software stack, developers look for languages that provide direct access to hardware, that have little runtime baggage, and that run both predictably and fast. Of course, that choice comes at a price. Since the entire code base of the operating system is usually written in a language with few or no built-in barriers for catching bugs such as memory errors, the entire code base is exposed to vulnerabilities.

Traditions should be challenged. Almost four decades ago, Ingalls [1981] proclaimed that operating systems can be dispensed with. This pronouncement came in the early days of Smalltalk. It can be understood as a part of a two-pronged argument. As Kell [2013] explains it, Ingalls was proposing to revisit the choice of implementation language; arguing that an operating system can be developed in a high-level, productivity-oriented programming language suitably extended with low-level hooks. This approach was followed by systems such as Bell Labs' Plan 9 [Pike et al. 1995] and Microsoft's Singularity [Hunt and Larus 2007] for example.

The other prong of Ingalls' argument is that the runtime system of a language like Smalltalk is already close to an operating system itself, and that there are benefits to be had by granting full control over the hardware to the language level. In this case, the operating system mostly disappears, shrinking down to a thin software layer in charge of dealing with the basic hardware conventions: interrupt handling, register management and bootstrapping. This approach was followed by Unikernels such as Mirage [Madhavapeddy et al. 2013] and bare-metal virtual machines such as FijiVM [Pizlo et al. 2009].

This paper explores the technical issues involved in fusing a programming language execution environment and an operating system into a single software artifact. We name the resulting blend a self-contained development environment. Our goal is to show how to minimize the low-level code that must be written for this blend to happen and to push services such as memory management, scheduling, and device drivers to the language level. While our paper focuses on implementation issues, the motivation for the work is our

Submitted, DLS, 2018

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

hope that some of the following advantages may be validated in the future:

Improved security: an operating system written in a managed language may expose fewer security vulnerabilities as the language has built-in error checking facilities.

Improved assurance: device drivers and protocols written in high-level languages may be easier to reason about, test and verify than a combination of C and assembly.

Educational use: the number of students familiar with productivity languages such as JavaScript is likely larger than those proficient in C. It may thus be possible to teach operating system construction mostly with higher-level languages (with only occasional dips into low-level code).

This paper does not attempt to validate the above hypotheses. Instead we focus on two artifacts: the NopSys framework and the CogNos operating system. The NopSys framework is a minimalist library for loading a language runtime into memory and forwarding hardware interrupts. It consists of two main components: 1) a loader that allows a system to bootstrap 2) event handlers that perform the essential register management required by the hardware and then delegates the actual handling to the language runtime. Overall, NopSys is made up by 3428 lines of C code and 608 lines of assembly code.

To validate that NopSys is sufficient to turn a programming language runtime into an operating system, we implemented CogNos, a bare-metal Smalltalk built using a lightly extended version of the Cog virtual machine [Miranda 2011] complemented with a Smalltalk implementation of device drivers for interrupt controllers, PCI controllers, IDE controllers, serial port devices, PS/2 mouse and keyboard, ATA hard disks, and a FAT32 file system. Consequently, CogNos is a reflective, live, and meta-circular Smalltalk development environment for x86 platforms. NopSys and CogNos are an evolution of the SqueakNOS project, which aimed to run Smalltalk on top of the predecessor of Cog virtual machine on the bare metal.

The main contribution of this paper is the design of NopSys and the validation that it suffices to implement a self-contained development environment. We have conducted a small number of experiments to evaluate the performance of CogNos, but these should be taken with a grain of salt. Operating systems are the sum of large development effort, carefully tuned over time. CogNos is a proof-of-concept system that was not optimized. We believe our results are encouraging, but there would certainly be much more work needed before one could consider deploying our system.

2 Background

The essence of an operating system is to interface with the computer hardware: the CPU, the memory, and the myriad of input/output devices. The landscape of operating systems is abundant and complex. There are operating systems covering many different software aspects, user requirements, and hardware architectures. For instance, mainstream general purpose operating systems such as Windows, Linux, or mac OS have almost nothing in common with real time operating systems targeted at embedded devices.

Most operating systems have a kernel, a single program running at all times. Moreover, hardware usually support two modes, a user mode and a privileged mode. The kernel is the only program that runs in privileged mode. To avoid the overheads of constantly polling devices, operating systems are interrupt driven. A programmable interrupt controller (PIC) interfaces between the myriad of devices and the CPU. The CPU checks for interruptions periodically and, depending on the priority of the current task and the interruption itself, it decides when to handle the interrupt. Several interrupts can be ignored (masked) by notifying the processor or the PIC.

2.1 Programming Language Virtual Machines

There are some similarities between the virtual machines used to implement languages such as Java, Python or Smalltalk, and operating systems. Virtual machines typically have a single kernel that is constantly running, and they may manage multiple users, often requiring some protection of the core of the system from user code. Virtual machines manage their memory and sometimes perform scheduling tasks over underlying resources. In fact, like an operating system, the purpose of a virtual machine is to shield application code from dealing with the rest of the platform on which it runs.

Cog is a virtual machine developed by Miranda [2011] and used, among other languages, by three different open-source Smalltalk releases: Cuis, Pharo [Black et al. 2009], and Squeak [Nierstrasz et al. 2009]. Cog is written in Slang, a restricted subset of Smalltalk which is compiled to C code. It features a just in time compiler, but also a bytecode interpreter.

The Cog VM can be extended with plugins, which add new primitive methods to the VM. Primitive methods form the API for hosted programs. They allow interaction with VM internals or the outside world. For example the primitive method `#basicNew` allocates a new object. Others exist to access files, create network sockets, etc.

3 NopSys – Pedal to the Metal

NopSys is a minimalist framework aiming to facilitate interfacing with x86 hardware. Its main features are processor initialization and hooks to register interrupt handlers. It

also manages build stages and debug configurations, usually cumbersome tasks in the context of complex systems. Applications which are statically linked against NopSys can be loaded by the GNU GRUB bootloader [Matzigkeit and Okuji 1999] and executed without an operating system. The resulting kernel can be booted on physical machines as well as virtualized hardware.

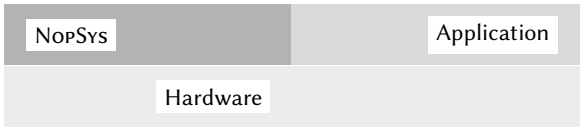

Figure 1. High-level architecture: NopSys is a library to run applications on the bare metal.

Figure 1 illustrates that NopSys is not a hardware abstraction layer (*i.e.*, a hypervisor) hosting applications. Instead, applications linked against NopSys become the lowest layer themselves. As such, NopSys has a wide range of possible applications. It could be used to:

- Run an arbitrary binary without the support of a standard operating system.
- Become the lowest layer of abstraction in library operating system approaches.
- Build systems where the operating system is fully replaced by services implemented in high-level programming languages.

To interface with the external world, applications need to interact with devices, hard disks, network cards, or the display. However, NopSys does not provide any hardware abstraction or drivers. This is not an oversight, but instead a tribute to its minimalist design. One particular use case that drove the design, was to simplify porting language virtual machines to run without the heavy-weight software stack of mainstream operating systems. The design we promote is to implement all the hardware support in a high-level programming language. Therefore the main goal for NopSys is to provide the support to bootstrap a language runtime. Everything else can be implemented in the high-level and memory safe language on top. This includes interrupt handling, memory management, device drivers, and hardware abstractions such as network protocols and filesystems.

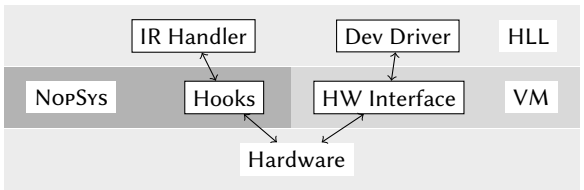

Figure 2. NopSys linked against a VM. The whole system, including interrupt (IR) handling and device drivers are implemented in the high-level language (HLL).

To corroborate NopSys usefulness we developed a system called CogNos. Like Figure 2 shows, we link an existing Smalltalk VM against NopSys and implement a significant amount of operating system services and drivers in Smalltalk. The resulting system features device drivers in a memory safe language by exploiting a minimal set of unsafe primitives. Given the inherent dynamic characteristics of Smalltalk, CogNos also enables to debug and change these drivers while the system is running. We come back to CogNos and describe it extensively in section 4. Now we describe NopSys' main features in more detail.

3.1 Bootstrapping

NopSys adheres to the mutliboot specification [Okuji et al. 2006]. Therefore it can be loaded by any compatible bootloader. We use GNU GRUB [Matzigkeit and Okuji 1999]. GRUB performs the basic video mode settings and initializes the CPU into 32-bit protected mode. It also loads the kernel files from disk into main memory and finally passes control to the NopSys `_loader` assembly routine. This routine sets up an identity-mapping paging scheme, passes the CPU into 64-bit long mode, and calls `nopsys_main`. Thereupon, in 64-bit long mode, the bootstrapping finishes by enabling SSE and setting up the basic interrupt handlers.

3.2 Interrupt Handling

On x86 there are three main sources of interruption: exceptions, hardware interrupts, and system calls. NopSys handles exceptions and hardware interrupts by implementing minimal routines that only save the execution context and delegate control to the application. NopSys currently does not support system calls.[1] All the system must be accessed through the corresponding language-level routine/method invocation model. Initially, all interrupts are masked. It is the responsibility of the NopSys client to communicate with the interrupt controller for unmasking the interrupts it needs to handle.

Listing 1 shows the six lines of assembly used to delegate interrupts to the corresponding NopSys client. The external function `signal_semaphore`, declared on Line 1, has to be implemented by the client. Line 2 saves the current context as mandated by the x86-64 application binary interface (ABI), line 3 places the interrupt number in register `rdi` and line 4 calls into the client's `signal_semaphore`. Line 5 restores the context and line 6 returns from the interrupt handling.

As the name `signal_semaphore` suggests, NopSys demands the client to have threads dedicated to handling interrupts. `signal_semaphore` then signals the corresponding client's semaphore and returns control to the interrupt handler. Accordingly, initial interrupt handling terminates instantly. When the client (language runtime) resumes execution it will

[1] In the current implementation there does not exist a kernel protection mechanism, so both application and kernel run on ring 0.

```
extern signal_semaphore
SAVE_CONTEXT
   mov rdi, %1  #Interrupt number
   call signal_semaphore
RESTORE_CONTEXT
iretq
```

Listing 1. NopSys interrupt handling

wake up the thread waiting at the corresponding semaphore. The actual reaction to the interrupt happens asynchronously, after control is transferred back to the client by `iretq`. It is the responsibility of the language-level handler to notify the PIC that the second stage of the interrupt handling has finished.

Hardware exceptions[2] such as general-protection faults, invalid opcodes, and division by zero are in most cases treated as fatal errors. This is usually not a problem because the application layer is expected to live in a managed environment, where program errors are caught early and handled safely. Finally, other exceptions, like page faults, require synchronous handling and are said to be continuable: they have to be handled immediately and return to the expression that originally caused the exception. In NopSys, continuable exceptions are handled like hardware interrupts, but by synchronously calling back into the client.

3.3 Runtime Support

A third ingredient of NopSys, not mentioned so far, is a minimal C runtime statically linked with the kernel. This part is mostly due to our aim to simplify linking language VMs with NopSys. Accordingly, NopSys includes a bare minimum subset of the C standard library [Ritchie et al. 1988], publicly known as libc, to work comfortably with VMs written in C. NopSys's libc provides support for I/O functions like `printf`, several string and math operations, and `malloc` for memory allocation. More powerful memory management support is expected to be implemented by the language VM. Most likely other eventual users of NopSys will expect to have this subset of the libc available too.

3.4 Build and Try Framework

The NopSys framework provides tools not only for programming, but also for building and debugging. This is accomplished by a series of Makefiles that include targets for different build settings to run the eventual clients on top of emulators, dynamic translators, and virtual machines with different kinds of debugging facilities. Currently, NopSys provides support for deploying to x86-64, Bochs [Lawton 1996], QEMU [Bellard 2005], Oracle VirtualBox, and VMWare VMPlayer.

[2]Interrupts 0 to 31

The NopSys build system takes as input a static object file with the compiled application and links it with its own code, creating an executable kernel. The following three requirements are necessary for clients to link against NopSys:

1. They must provide a `nopsys_vm_main` entrypoint.
2. They must not be linked with the standard C library (using -nostdlib gcc flag at compile time), as NopSys provides its own implementation.
3. They must be compiled without using a red zone for the stack (using -mno-red-zone). Otherwise, interrupt handling routines might corrupt the C stack because the client will run in ring 0.

Finally, the NopSys build system supports creating a bootable disk image which includes the GRUB bootloader. We provide options to build ISO 9660 or hard disk images. It is also possible to choose a hardware virtual machine appliance and to automatically launch an instance. Furthermore, we include facilities to launch some of those virtual machines with a gdb stub to remotely debug the resulting system.

4 CogNos – A Self-Contained Smalltalk

As a proof of concept of the NopSys framework we developed CogNos, a a Pharo Smalltalk image running on top of Cog VM with NopSys extensions. Device drivers and operating system services are implemented at the Smalltalk level. CogNos allows us to explore supporting *close-to-the-hardware* services in a high-level, object-oriented and live environment. It consists of two main parts:

1. A plugin extending the Cog VM with support for interfacing with the hardware.
2. Smalltalk-level packages providing operating system services.

CogNos runs on the bare-metal and relies on the NopSys framework for the most basic hardware management (see Figure 2). The main role of the VM plugin is to expose hardware access to Smalltalk programs, partially following the high-level low-level programming approach described by Frampton et al. [2009]. The Smalltalk packages implement interrupt handlers, provide basic device drivers, and add support for common operating system abstractions such as filesystems and network protocols.

Unlike most operating systems, CogNos is written entirely in a high-level language. For instance, consider the software layers involved when saving a file: the FAT32 [Microsoft 2000] filesystem, a block device driver, the ATA [Group 2003] bus protocol, direct memory access (DMA), PCI and PIC drivers. Each of those is provided by Smalltalk packages. Accordingly, given the dynamic nature of Smalltalk, they can be debugged, modified, and extended inside a live system.

The reminder of this section first describes the Cog VM plugin and then explains the operating system services implemented in Smalltalk. An evaluation to measure the performance overhead of this high-level system programming style is presented in section 5.

4.1 Cog VM Plugin

As most of the Cog VM, CogNos' plugin is written in Slang. To minimize the low-level language code and promote developing as much as possible at the language-level, the plugin seeks a minimal design approach. It consists of about 25 methods of which 5 are in charge of debugging information. The rest of the methods extend the Cog VM with *unsafe* operations necessary to implement different kinds of hardware support in Smalltalk.

Four methods provide support for reading and writing the corresponding CPU control registers (`cr 0–3`). These registers manage the execution modes and the memory layout settings of the platform. One method reads the time stamp control register (`rdtsc`) needed for measuring time. Twelve of the remaining methods provide support for interacting with the hardware I/O ports, which provide a direct communication with devices. They are twelve because of the combination of reading/writing operations with different amounts of information (short, word, double word). Finally, one method to bind a Smalltalk semaphore to a NopSys interrupt handler.

Listing 2 illustrates the implementation of the plugin by showing two primitives: the first in charge of reading the value of the control register `cr0` and the latter in charge of binding a semaphore, created at the language-level, with a low-level NopSys interrupt service routine. The code was simplified by removing the error handling behavior. For reading `cr0` lines 3 and 4 initialize variables while line 5 loads the corresponding register into the answer variable. Line 6 converts the value into an Smalltalk integer, and line 7 pushes it to on the Smalltalk VM stack. The implementation for the remaining control registers is analogous. For the twelve I/O primitives the implementation is similar. The main difference is that they use the `in`/`out` inline assembly instruction with their corresponding parameter management.

For the second primitive, line 12 declares variables, and line 13 tells the compiler that the code will use the NopSys array of `irq_semaphores`. Lines 14 and 15 read the parameters from the Smalltalk stack. Line 16 finally sets the Smalltalk semaphore identifier into the corresponding index. This essentially tells NopSys that, whenever it receives a notification that the corresponding interrupt was signalled (see subsection 3.2), the Cog VM must wake up the respective semaphore. Lines 17 and 18 again deal with passing the result back to the Smalltalk VM. In this case the result is always `true` because we omit the error handling. Note that, overall, `primitiveRegisterSemaphoreIndexForIRQ` enables to create or modify an interrupt handler at the language-level and at run-time.

```
1  primitiveReadRegisterCr0(void)
2  {
3     unsigned long answer;
4     int return_value;
5     asm("movq %%cr0, %0" : "=a" (answer) );
6     return_value = positive32BitInt(answer);
7     popthenPush(1, return_value);
8  }
9
10 primitiveRegisterSemaphoreIndexForIRQ(void)
11 {
12    int semaphore_id, irq_number, return_value;
13    extern irq_semaphores[];
14    semaphore_id = stackIntegerValue(1);
15    irq_number = stackIntegerValue(0);
16    irq_semaphores[irq_number] = semaphore_id;
17    return_value = trueObject();
18    popthenPush(3, return_value);
19 }
```

Listing 2. Two Slang primitives from the CogNos plugin already translated to C.

4.2 OS Library

The biggest portion of CogNos is written directly in Smalltalk. The code provides device drivers for accessing CMOS memory, PS2 keyboard and mouse, 16550 UART serial circuits, the PCI bus, the Intel 8259 programmable interrupt controller, the Realtek 8139 PCI ethernet card, and ATA controllers. It also provides support for FAT32 filesystems, and a full TCP/IP network stack is ongoing. For statistics on the size of the code we redirect the reader to section 5.5.

To illustrate the packages we exhibit code snippets that describe behaviors at different abstraction levels. We start by describing the `initialize` method of the `Computer class` in Listing 3. The Smalltalk image running on top of CogNos is configured such that it activates this method during its startup phase. The method initializes a processor and a dictionary of devices in Lines 2 and 3. Then, it installs a series of devices. Each device registers to the devices dictionary during its initialization step. Finally, a filesystem is initialized.

Each device provides its own initialization depending on the particular hardware it manages. As an example, Listing 4 shows the installation of the keyboard. Line 3 instantiates the keyboard on port 96. This keyboard object is the handler that will finally be signalled by the NopSys interrupt service routines described in Listing 1. Line 4 registers it as the interrupt handler for IRQ 1. From line 6 on, the code shows how CogNos handles keyboard interrupts. In line 8 the code reads from the keyboard the raw bytes representing the event.

```
Computer>>initialize
  processor ← X86Processor new.
  devices ← Dictionary new.
  PIC8259 installOn: self. "interrupt controller"
  PS2Keyboard installOn: self.
  ATAController primary installOn: self.
  primaryFilesystem ← self diskFS.
```

Listing 3. Computer initialization

```
PS2Keyboard class>>installOn: aPC
  | keyboard |
  keyboard ← self onPort: 16r60.
  aPC pic addHandler: keyboard forIRQ: 1.

PS2Keyboard>>handleKeyboardIRQ
  | scanCode key |
  scanCode ← self readKeyboardData.
  decoder nextScanCode: scanCode.
  decoder isModifier ifFalse: [ "ascii char"
    key ← decoder keyUsing: keysMapping
     modifiers: modifiersAndButtons.
    decoder isKeyUp ifTrue:
      [ self newKeyboardUpEvent: key ] ifFalse:
      [ self newKeyboardDownEvent: key]]
  ifTrue: [ " Code when being a modifier... "].
```

Listing 4. Computer initialization

Then it decodes exactly what kind of key has been pressed and acts accordingly.

To conclude the explanation of the Smalltalk packages, Listing 5 describes a method of the FAT32 Filesystem implementation in charge of returning the files contained in a directory. Essentially, the method looks for the clusters of the block device in which the directory has been stored. Then it goes through all of them and collects the file records they contain, to finally return the list of files.

4.3 C Code

The Cog VM is written in a mixture of Slang and C code. Slang is translated into C code, after which all code is compiled with a C compiler. The code is split in OS-independent and OS-dependent code. CogNos adds to Cog a new platform, NopSys, to implement the minimum interface that Cog VM expects from any OS. This includes the following platform-specific functionality:

Image reading The image is loaded into memory by GRUB. The CogNos main procedure receives the start address from NopSys.

Memory layout NopSys finds the last memory address used by the kernel by relying on the multiboot specification interface implemented by GRUB. CogNos manages memory after the Smalltalk image has been initialized. The heap can then grow freely towards the end of memory.

Timing information management CogNos uses the NopSys functions provided for reading the timestamp counter and for converting clock ticks to microseconds.

Compatibility Other functionality that is implemented as empty stubs, like accessing the native operating system clipboard.

```
FAT32Filesystem>>filesFor: aDirectory
  | cluster files clusters subfiles |
  files ← OrderedCollection new.
  clusters ← self clustersChainFor: aDirectory.
  clusters do: [ :aNumber |
    cl ← self clusterAt: aNumber.
    subfiles ← cl fileRecords collect: [ :rec |
      NOSFile
        named: record name
        identifier: record firstCluster
        size: record size ].
    files addAll: subfiles ].
  ↑ files
```

Listing 5. Computer initialization

As the final step the CogNos Makefile includes a call to the previously described NopSys Makefile, which takes the CogNos object file, links everything together and enables to launch (and eventually debug) CogNos on top of hardware virtual machine.

5 Evaluation

In this section we report on a series of experiments[3] aiming to provide a general overview of the impacts the NopSys approach carries in terms of performance and code size. We compare the performance of Smalltalk benchmarks as executed by Cog on Ubuntu OS against CogNos. Then, we measure the impact of performing I/O in CogNos. Finally we compare boot times and the difference in terms of code sizes and memory footprints.

5.1 Subjects

To perform the experimentation we use three different operating systems to compare with depending on the experiment: MINIX 3 [Tanenbaum and Woodhull 1987], Ubuntu/Desktop, and Ubuntu/Server. MINIX 3 is a free open-source operating system featuring a micro-kernel and supporting a wide range of hardware devices and software protocols for both, ARM and x86 platforms. It is mostly written in C and aims at achieving high reliability through fault tolerance and

[3] In https://github.com/nopsys/CogNOSExperiments the interested reader will find instructions on how to repeat and reproduce the experiments.

self-healing techniques. To do so, most of MINIX 3 features (including the memory manager) run as server processes at the user level. Because of its size and approach it is well suited for both embedded and standard appliances. Being a well documented operating system, suitable for research as well as for industrial applications, in both embedded and standard architectures, and with a minimal kernel approach, we consider MINIX 3 provides a fair baseline for comparison against CogNos.

On the other hand, Ubuntu is one of the world's most-popular multi-platform open-source operating systems, covering a wide range of use cases. Both versions, desktop and server, feature the same kernel. The main difference is that the server version runs in headless mode and does not include several general purpose applications such as the office suite, web browsers, and multimedia like software.

As Smalltalk benchmarks we chose the suite proposed by Marr et al. [2016] for cross-comparing dynamic language implementations. The benchmarks were collected from various sources such as Computer Language Benchmarks[4], Octane[5], as well as independent additions. Their size ranges from 2 to 20 classes, and 20 to 350 lines of executed code and are mostly computation bound. They do not perform any kind of I/O. To account for the non-determinism in modern systems, each benchmark result is based on 10 in-process iterations and execution time is measured after 10 iterations, which are counted as warmup.

To run all the experiments under the same setting we resort to install the aforementioned operating systems and benchmarks on top of a VirtualBox bundle with a single processor and 2GB of RAM. The machine running the virtual environment and collecting the results is a quad-core Intel Core i7-3770, 3.50 GHz with 16 GB RAM, running OS X High Sierra version 10.13.5.

5.2 Language-Level Performance

To understand the performance impact of implementing operating system services in a high-level language we compare a set of benchmarks in CogNos with a standard Cog VM on top of Ubuntu Desktop. To avoid inaccuracies because of different artifact versions we use in both cases a VM compiled from the same source code and a very similar set of activated plugins. The only difference is that the VM for Ubuntu is compiled along with a couple of plugins needed for running Cog under Unix-like environments while the CogNos VM includes the CogNos plugin (see Section 4.1). The Smalltalk image is the same but it runs a slightly different initialization process depending on the environment under which it is running. For each measurement we report performance with the JIT compiler enabled and disabled.

[4]The Computer Language Benchmarks Game, Isaac Gouy, https://benchmarksgame-team.pages.debian.net/benchmarksgame/

[5]Octane, Google, https://developers.google.com/octane/

Table 1. Overall Baseline Results

Runtime	OF	CI-95%	Sd.	Min	Max	Median
CogNos-Jit	0.97	<0.88 - 1.06>	0.13	0.64	1.14	0.99
CogNos-Int	1.12	<0.87 - 1.44>	0.64	0.85	2.85	0.95

Figure 3 shows boxplots of the results for both, the JIT and the interpreter versions of CogNos. Each boxplot represents the results normalized to the mean of the benchmarks running Cog on Ubuntu for each respective mode. It can be seen that for both versions, the boxplots fall almost entirely around 1x±0.15. Table 1 presents these results along with several statistical values such as the confidence interval for each of the results. The mean overhead of the JIT version is 0.97 while the interpreter runs 1.12 times slower respectively.

Figure 4 presents the results for each individual benchmark. The top part of the figure shows overheads for CogNos in interpreted mode. The bottom part is analogous but with the JIT enabled. In the interpreted case the benchmarks are usually at the same level of performance in both settings but Mandelbrot and NBody have a significant overhead in CogNos (around 3x and 2x slower respectively). These two benchmarks explain why considering the mean of all of the benchmarks, CogNos interpreter is around 12% slower than Cog. GC *TODO: Address rev.2 concern: In the interpreter evaluation, only the benchmarks heavy on double arithmetic are considerably slower. Can you explain why? it looks like the C compiler generate different machine instructions than the JIT and that those instructions are not well supported in your context (Typically, C compiler tends to generate AVX instructions instead of SSE2 these days, and likely the Cog's JIT generates SSE2 instructions). But I can't tell for sure. In general I miss explanations in the section, you say it's x times slower or x times faster, but you do not explain why.*

When analyzing the results for the JIT compiler version, again most of the benchmarks perform evenly but DeltaBlue runs significantly faster in CogNos. We noticed that this happens mainly because DeltaBlue has a long compilation phase that finishes faster in CogNos. Our preliminary observations show that most of the benchmarks reach the end of the compilation phase sooner under CogNos. The worst overheads of CogNos comparing the JIT versions are below 20% (Storage and List).

Overall, the results suggest that running a system on top of NopSys does not significantly impact performance for computation-based scenarios, *i.e.*, scenarios not performing input/output operations. In addition we observed two interesting facts: the benchmarks in CogNos appear to be more predictable (the boxplots are smaller) and the warming up phase of the JIT compiler seems to be faster. Although we still need to perform more exhaustive experiments, we assume the overall lower system load on CogNos, can explain both of these effects.

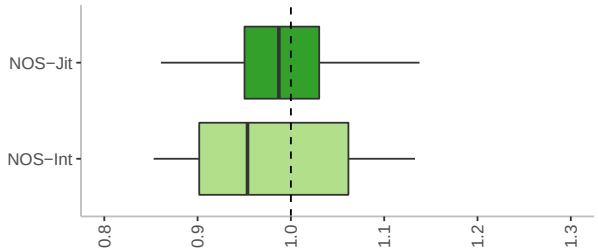

Figure 3. Overall overhead factor of 12 benchmarks on CogNos vs Cog with Ubuntu 17.10 for both, interpreted and JIT versions. Benchmarks were selected from Marr et al. [2016].

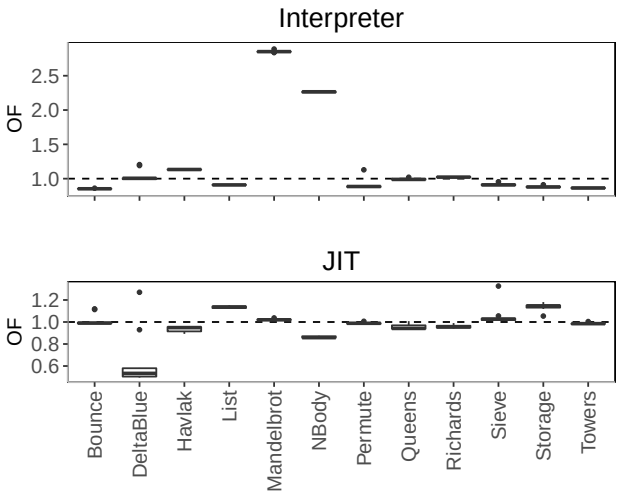

Figure 4. Overhead factor (OF) of NopSys vs Cog/Ubuntu17.10 for both, interpreted mode (above) and JIT mode (below). Baselines are an interpreter and a JIT version of Cog respectively. Each experiment is executed within VirtualBox.

5.3 Hard Disk and Filesystem Performance

This experiment intends to give an idea of the overhead incurred by I/O bound applications, *i.e.*, applications demanding a considerable amount of input/output operations. To do so, we design an ad-hoc benchmark that walks through all the files of a directory and reads their content. We measure the time it takes to run the benchmark under three different settings. First, CogNos using its own set of software layers for performing all the operations. In addition, we also measure CogNos but using a ramdisk, *i.e.*, a FAT32 mounted directly from memory. Finally, we assess Cog running on top of Ubuntu. The filesystem for the CogNos version is *FAT32*, the only filesystem CogNos currently supports. For Cog under Ubuntu, we decide to keep its standard filesystem, *ext4*. The directory contains 100 files, each containing only one character.

Figure 5 shows the results in logarithmic scale. Table 2 shows the final results along the corresponding statistical

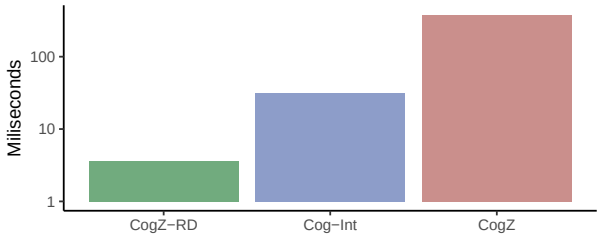

Figure 5. Time it takes to CogNos vs Cog/Ubuntu17.10 to read 100 files from the same directory. CogNos-RD uses a FAT32 filesystem mapped to a ram disk, CogNos uses a FAT32 filesystem mapped to the ATA controller, and Cog uses an ext4 filesystem. All versions use the interpreted VM flavor.

Table 2. Overall Filesystem Results

Runtime	Time	CI-95%	Sd.	Median
CogNos-RD	3.65	<3.42 - 3.87>	0.61	5.14
Cog	31.23	<30.63 - 31.84>	1.62	36.14
CogNos	381.62	<376.32 - 386.93>	14.20	418.47

values. Our current implementation is around 10x times slower than Cog under Ubuntu for this I/O driven benchmark. But, when using the ramdisk (CogNos-RD) instead of disk access, the results are 100x faster, exposing the fact that most of the overhead is caused by the disk access. It is worth noting that other library operating systems reach I/O performance on par with standard operating systems under similar conditions [Engler et al. 1995; Madhavapeddy et al. 2013]. Consequently, these results expose that there is still a significant need to improve our Smalltalk hardware management services in order to reduce the current excessive input/output overheads.

5.4 Boot Time

We now measure the time it takes CogNos, Ubuntu desktop (UbuD), Ubuntu server (UbuS), and Minix3 to boot up. To be as precise as possible we report the raw cycles since the systems are powered on along with the cycles normalized to the minimum value of the sample. For CogNos we report the cycles until the Smalltalk initialization phase finishes. All the device drivers initialization takes place within that phase as explained in section 4.2. For Minix3 we report the cycles until the multi-user part of the init process (which mainly loads the filesystem and performs some basic initialization) ends. For each Ubuntu we report the time until the initialization of the first service.

Since CogNos is the fastest of the compared systems we use it as baseline. Figure 6 shows the respective overheads of the other systems. Table 3 shows the final numbers. CogNos boots in 10^9 less cycles than MINIX3, which is the most fair

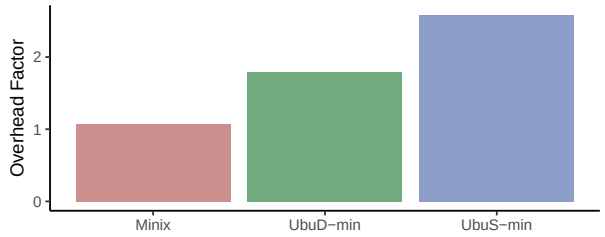

Figure 6. Boot times comparison from the moment the system is powered on normalized to the amount of cycles it takes CogNos to boot up. Ubuntu boot times are measured up to the point where the first service is initialized.

Table 3. Boot Results

VM	Cycles	OF
CogNos	1.37e+10	1.00
Minix	1.47e+10	1.07
UbuD-min	2.47e+10	1.79
UbuS-min	3.56e+10	2.59

of our comparisons. Accordingly, Minix boots around 1.07x slower than CogNos. On the other hand Ubuntu and Ubuntu server versions take in the order of 10^{10} (1.8x) and $2 * 10^{10}$ (2.6x) more cycles respectively to start the init process.

CogNos does very few things for booting up. Essentially, it only needs NopSys to load the Smalltalk image in memory and initialize the processor registers. After that, it immediately starts running the Smalltalk image. Within Smalltalk, CogNos mainly enables the needed hardware and registers the corresponding interrupt handlers. The system does not need to deal with different users, login systems, nor complex memory management schemes. Accordingly the results are expected.

5.5 Source Code Statistics

Table 4 shows statistics about the whole codebase of both NopSys and CogNos. NopSys consists of 608 Assembly LOC and 3428 C LOC. The assembly code mainly defines all the needed interrupt handlers (see section 3.2). In additions, it also deals with the initialization of basic structures needed by hardware convention such as the global descriptor table. It also provides support for managing several CPU registers such as the *control registers* and the *tsc* (time stamp counter) needed for benchmarking.

The C code deals with the initialization of structures at the C level, implements several functions for managing the display using a *framebuffer*, and defines several general purpose functions from the libc. In addition, it also includes development tools such as a serial device driver for receiving debug information when the display is not working and a debug console in which to print any kind of messages. Hence,

Table 4. Source code statistics

Package	Language	Classes	Methods	LOC
Source code	Assembly	1	0	608
Libc headers	C	1	0	90
Source code	C	1	0	1,461
Headers	C	1	0	1,877
Dev-Base	Smalltalk	9	422	1,631
Dev-Processor	Smalltalk	2	8	25
Dev-Storage	Smalltalk	6	153	865
Ext-Structures	Smalltalk	11	135	419
Filesystems	Smalltalk	4	57	206
FAT32	Smalltalk	7	197	900
Kernel	Smalltalk	5	117	439
Storage	Smalltalk	19	180	718

it is worth noting that from the 3428 C LOC, about 294 are for supporting debugging tools and 1562 encode the fonts used by the system. Consequently, this leaves out only 1572 C LOC describing essential behavior.

Now, considering in addition CogNos, as Table 5 summarizes, the code also contains 5203 lines of Smalltalk code. This includes all the device drivers, software protocols and primitive operations fully described in section 4.2. Overall, CogNos and NopSys contains in the order of 10000 LOC at the moment of this writing. On the other hand, if we take into account also the whole sources from a Pharo Smalltalk image, there are 653746 Smalltalk LOC. This includes many classes and libraries which are not needed by nor used by CogNos. The Cog/JIT VM source code is made of around 430000 lines of C code (including C translated from Slang) and 10000 assembly lines.

The MINIX 3 microkernel is only about 12,000 lines of C and 1400 lines of assembler for very low-level functions such as catching interrupts and switching processes [Tanenbaum and Bos 2014]. For a matter of completeness we also provide basic statistics about the whole project. At the time of writing, the official MINIX3 repository [Minix 2018] reported 7,5 million executable LOC mostly written in C and C++. This includes a heterogeneous set of device drivers, software protocol implementations, and even package manager applications. Finally, according to the Linux foundation official reports [LinuxFoundation 2017], the 4.13 version of Linux kernel already contains about 25 millions of LOC.

5.6 Discussion

In terms of inherent performance, at least for computational driven applications, the benchmarks expose that the overheads are in general low, if there are any. Only two benchmarks exposed excessive overheads under the interpreted

Table 5. Source code overall

Language	Classes	Methods	LOC
Assembly	1	0	608
C	3	0	3,428
Smalltalk	63	1,269	5,203
Total	67	1,269	9,239

version. Both intensively use floating point arithmetic operations.

The hard disk experiment challenges exhaustively CogNos capabilities. The results showed that CogNos still needs important performance improvements to become practical under I/O driven scenarios. Since our main goal was to demonstrate its feasibility, in most of the development choices we opted for a naive approach instead of complex but performant algorithms and data structures. This is exactly the opposite of mainstream operating systems like Linux based ones. For instance the filesystem layer in Linux is heavily optimized with ad-hoc data structures and caching systems at different levels.

Finally, in the case of boot loading time CogNos outperformed MINIX 3. We also illustrate that CogNos code size is relatively small in comparison to MINIX 3. For a matter of completeness we also show that mainstream general purpose operating systems like Linux have a considerably larger code base. Nevertheless, it is worth noting that our comparisons in this experiment are only informative. It is not fair to compare the size of general purpose, multi-user, stable, and portable operating systems with our research proof of concept implementation.

Threats to validity. NopSys is a language-independent framework. However, we only assessed it against the Cog VM. Testing the library with more runtimes present considerable challenges. On one hand, the runtime must implement the NopSys interface. Although we carefully designed the library so that this task is simple, language runtimes are complex software artifacts. On the other hand, to make it actually usable, the approach needs a language-level implementation of several device drivers and software protocols. The mitigation of this threat relies on the fact that NopSys is minimalist, and revisiting its source code shows that there is no language dependent code.

Regarding the performance results, the set of benchmarks that we selected are not representative of general applications. Accordingly, the results may differ under user particular scenarios. To partially mitigate these threats, we measure times for computational based benchmarks as well as for I/O based applications. Most of the benchmarks were already used to measure Smalltalk implementations. Moreover, we decided to use VirtualBox for running the experiments to setup the whole benchmarking within the same machine. This additional layer may influence the results.

6 Related Work

Smalltalk-80 [Goldberg and Robson 1983] descendants, and languages following Smalltalk's tradition, like Self [Ungar and Smith 1987], were originally conceived to be self-contained. For instance, they provide their own user interfaces and the languages only interact with a runtime providing a significant amount of low-level services. We are not aware of implementations of these languages that run on the bare metal or implement their own device drivers. In contrast CogNos not only run on the bare metal but pushes, as far as possible, the interaction with the hardware to the language level.

A number of languages were extended to act as self-contained environments. Mesa programming language [Mitchell et al. 1978] was a precursor to Modula-2 [Wirth 2012], which served as the programming language for both the applications and the operating system of Lilith [Knudsen 1983]. Oberon [Wirth 1989], a successor to Modula-2, was used to implement the Oberon operating system. The first Java operating system was JavaOS from Sun [Saulpaugh and Mirho 1999]. JavaOS relied on an interpreter and was not widely adopted. It was followed by JX [Golm et al. 2002] which ran on x86 and PPC and supported dynamic compilation. The authors reported slowdowns of about 50% in comparison to Java on Unix. One notable feature of JX was its support for component isolation to improve security and isolate failure points. The latest effort in this direction is FijiVM, a Java virtual machine for embedded and real-time system that runs on bare metal [Pizlo et al. 2010]. While FijiVM had a larger C code base than CogNos, it also allows developers to write interrupt handlers and device drivers in Java [Pizlo et al. 2009].

In the functional setting, Hallgren et al. [2005] presented a restricted monadic framework to access low-level hardware features aiming at safety. Their work describes the challenges of implementing the framework as well as attempts to formalize it. As a proof of concept the authors describe House, a small operating system coded almost entirely in Haskell using the monadic interface for the IA32. The core of Haskell is type-safe and memory-safe, which prevents many classes of bugs, and it is also pure, which eases reasoning about program behavior.

Our work differs from the aforementioned in two main different aspects. First, NopSys is a language-agnostic library. Second, our proof of concept is a Smalltalk system, which had not been explored before. This means that we had to support a graphical user interface and enable a live-programming experience for system development.

Monolithic operating systems run all services in kernel mode and interact with user space via system calls. In contrast, microkernels seek to define a minimal subset of essential services and leave all other functionalities to the user level. For instance, this may mean to provide only scheduling, memory management, and inter-process communication. Our approach with NopSys is kernel-agnostic. A Smalltalk system naturally tends toward the monolithic side of the equation as it is a single-user, everything shared system. One can easily conceive using NopSys together with a language that provides isolation and then use the language's mechanisms for protection. This was in essence the approach taken in Hunt and Larus [2007]. In Java, the work on Isolates demonstrated how to achieve isolation [Czajkowski 2000] and how to benefit from properties of the language for efficiency [Palacz et al. 2002].

Exokernels [Engler et al. 1995] propose to provide application-level management of physical resources, as well as inter process communication and memory management. Their main hypothesis is that the rigid interface imposed by the operating system to hardware limits the performance and implementation freedom of applications. The exokernel acts as a multiplexer of available hardware and provides resources protection. The authors view the operating system as a library in charge of implementing all the rest of the behavior. The NopSys approach facilitates the development of library operating systems.

Unikernels [Madhavapeddy et al. 2013] are another example of library operating systems. Unikernels enable users to write applications in high-level languages for then specializing them into a standalone kernel at compile time. The unikernels foundational paper presents Mirage, a unikernel for OCaml written applications targeting commodity clouds. They manage to provide very small appliances (200KB) with better performance results than Unix appliances (400Mb).

In contrast to NopSys, Mirage still implements a low-level interface for the most basic device management. For instance, to avoid portability issues it relies on the Xen hypervisor device model. Moreover, Mirage was designed for converting applications target to the commodity cloud into freestanding while NopSys aims at achieving so for general purpose applications. Finally, OCaml is a statically typed programming language.

7 Conclusions

This paper presented NopSys, a thin software layer that provides the minimal services needed to let a language runtime execute without an operating system. To validate our contention, we turned the Cog VM, a Smalltalk virtual machine, into a single-user application running on the bare-metal. The result is CogNos, a self-describing and reflective programming environment in charge of managing the whole system: from the very basic interrupt handling up to the software abstractions for realizing input/output such as a FAT32 filesystem. We finally present a preliminary empirical evaluation that suggests that the approach is viable.

The next steps in this research direction include validating that the advantages we claimed in the introduction hold. Going forward we may select different programming languages as vehicles of investigation, or stick with Smalltalk. For improved security, it would be interesting to attempt to define a minimal kernel that can be formally validated. For improved assurance, it would be interesting also to compare the effort involved in verifying a device driver written in a low-level language, with that of one written, in say, Smalltalk. Lastly in terms of education, we would be interested in developing a full-fledged research operating system and compare the level of effort of writing different components in Smalltalk versus C.

Acknowledgments. This work received funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (grant agreement 695412), the NSF (award 1544542 and award 1518-844) as well as ONR (award 503353).

References

Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator.. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.

Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. 2009. *Pharo by Example.* Square Bracket Associates. http://pharobyexample.org

Microsoft Corporation. 2000. *Microsoft Extensible Firmware Initiative FAT32 File System Specification.* Technical Report. http://download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/fatgen103.doc

Grzegorz Czajkowski. 2000. Application Isolation in the Java Virtual Machine. In *Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA).* https://doi.org/10.1145/353171.353195

D. Engler, M. Kaashoek, and J. Toole 1995. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Symposium on Operating Systems Principles (SOSP).* https://doi.org/10.1145/224056.224076

Daniel Frampton, Stephen Blackburn, Perry Cheng, Robin Garner, David Grove, J. Eliot Moss, and Sergey Salishev. 2009. Demystifying Magic: High-level Low-level Programming. In *International Conference on Virtual Execution Environments (VEE).* https://doi.org/10.1145/1508293.1508305

Adele Goldberg and David Robson. 1983. *Smalltalk-80: the language and its implementation.* Addison-Wesley

Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. 2002. The JX Operating System. In *USENIX Annual Technical Conference (ATC).* http://dl.acm.org/citation.cfm?id=647057.713870

Serial ATA Working Group. 2003. *"Serial ATA: High Speed Serialized AT Attachment.* Technical Report. http://www.serialata.org

Thomas Hallgren, Mark Jones, Rebekah Leslie, and Andrew Tolmach. 2005. A Principled Approach to Operating System Construction in Haskell. In *International Conference on Functional Programming (ICFP).* https://doi.org/10.1145/1086365.1086380

Galen Hunt and Jim Larus. 2007. Singularity: Rethinking the Software Stack. *Operating Systems Review* 41 (April 2007), https://doi.org/10.1145/1243418.1243424

Dan Ingalls. 1981. Design Principles Behind Smalltalk. *BYTE Magazine* (Aug. 1981), 286–298.

Stephen Kell. 2013. The Operating System: Should There Be One?. In *Workshop on Programming Languages and Operating Systems (PLOS)*. https://doi.org/10.1145/2525528.2525534

S.E. Knudsen. 1983. *Medos 2, a Modula 2 Oriented Operating System for the Personal Computer Lilith*. https://books.google.com.ar/books?id=sSYBjwEACAAJ

Kevin P Lawton. 1996. Bochs: A portable pc emulator for Unix/x. *Linux Journal* 1996, 29es (1996).

LinuxFoundation 2017. LinuxFoundation. https://www.linuxfoundation.org/2017-linux-kernel-report-landing-page/.

Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/2451116.2451167

Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-language Compiler Benchmarking: Are We Fast Yet?. In *Dynamic Languages Symposium (DLS)*. https://doi.org/10.1145/2989225.2989232

Gordon Matzigkeit and Yoshinori K Okuji. 1999. the GNU GRUB manual.

Minix 2018. Minix Repository. https://www.openhub.net/p/minix3

Eliot Miranda. 2011. The Cog Smalltalk Virtual Machine. In *Workshop on Virtual Machines and Intermediate Languages (VMIL)*.

James G Mitchell, William Maybury, and Richard E Sweet. 1978. *Mesa language manual*. Technical Report. Xerox Res. Cent.

Oscar Nierstrasz, Stéphan Ducasse, and Damien Pollet. 2009. *Squeak by example*. Lulu.com.

Yoshinori K Okuji, Bryan Ford, Erich Stefan Boleyn, and Kunihiro Ishiguro. 2006. The multiboot specification. *Version 0.6* 95 (2006), 173.

Krzysztof Palacz, Jan Vitek, Grzegorz Czajkowski, and Laurent Daynès. 2002. Incommunicado: efficient communication for Isolates. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. https://doi.org/10.1145/582419.582444

Rob Pike, David L. Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. 1995. Plan 9 from Bell Labs. *Computing Systems* 8, 2 (1995), 221–254.

Filip Pizlo, Lukasz Ziarek, Ethan Blanton, Petr Maj, and Jan Vitek. 2010. High-level Programming of Embedded Hard Real-Time Devices. In *EuroSys Conference*.

Filip Pizlo, Lukasz Ziarek, and Jan Vitek. 2009. Real time Java on resource-constrained platforms with Fiji VM. In *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*. 110–119.

Dennis M Ritchie, Brian W Kernighan, and Michael E Lesk. 1988. *The C programming language*. Prentice Hall Englewood Cliffs.

Tom Saulpaugh and Charles A Mirho. 1999. *Inside the JavaOS operating system*. Addison-Wesley.

Andrew S. Tanenbaum and Herbert Bos. 2014. *Modern Operating Systems* (4th ed.). Prentice Hall.

Andrew S Tanenbaum and Albert S Woodhull. 1987. *Operating systems: design and implementation*. Vol. 2. Prentice-Hall.

David Ungar and Randall B. Smith. 1987. Self: The Power of Simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications (OOPSLA)*. http://doi.acm.org/10.1145/38765.38828

Niklaus Wirth. 1989. From Modula to Oberon: The Programming Language Oberon. *ETH, Eidgenössische Technische Hochschule Zürich* 111 (1989).

Niklaus Wirth. 2012. *Programming in MODULA-2*. Springer Science & Business Media.