Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Rikke Solbjørg

Evaluating Code Overlays on the Oberon System Using RISC-V

Master's thesis in Computer Science
Supervisor: Michael Engel
July 2021

NTNU
Norwegian University of
Science and Technology

Rikke Solbjørg

Evaluating Code Overlays on the Oberon System Using RISC-V

Master's thesis in Computer Science
Supervisor: Michael Engel
July 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

Code overlays are a versatile technique to reduce the memory footprint of programs, by allowing parts of the program to be loaded from secondary storage into memory as they are needed. While it is an old technique, it has been seeing renewed interest in the context of embedded RISC-V systems without memory management units.

This thesis evaluates automatically generated code overlays on RISC-V in the context of an operating system, Project Oberon 2013. A prototype for an overlay system on the Oberon system is developed, and along with it several different strategies, including one that incorporates Oberon's heap allocation and garbage collection mechanisms.

The thesis concludes by evaluating the benefits and drawbacks of the presented strategies in the context of Pareto optimality. It finds that while one strategy is able to minimise the memory footprint substantially, it can only do this at the cost of considerable performance degradation. Another strategy, utilising the system heap, is able to strike a balance between performance and minimising the memory footprint of the system's code.

Sammendrag

Code overlays er en teknikk for å redusere minnet brukt av programkode ved å laste deler av programmet fra sekundærminnet til primærminnet etter behov. Til tross for å være en gammel teknikk har det nylig vekt interesse til bruk i enkle RISC-V-systemer som mangler minnehåndterere (MMU).

Denne masteroppgaven vurderer bruken av automatisk genererte code overlays til RISC-V, under operativsystemet Project Oberon 2013. En prototype av et slikt overlay-system utvikles, samt diverse strategier. En av disse strategiene inkorporerer operativsystemets innebygde systemer for heap allocation og garbage collection.

Masteroppgaven konkluderer ved å vurdere strategiene opp mot hverandre ved bruk av Pareto-optimalitet. Denne analysen viser at dog én strategi evner å merkverdig redusere systemets totale minnebruk, klarer den kun dette ved en drastisk påvirkning på systemets totale ytelse. En annen strategi, som bruker systemets heap, finner en mer ønskelig balanse mellom minnebruk og ytelse.

Acknowledgements

Writing a master's thesis is a long and difficult process, even moreso after a year of living with a pandemic. These last six months have been both amazing and stressful, and I couldn't have done it without the help of all the people around me.

I owe many thanks to my advisor, Michael Engel, for supporting me with insights, papers, and lots of support when I needed it. Thank you for helping me with this project, for reminding me of the purpose of my work, for believing that I could pull through even when I was stuck, and for providing your ideas as well as building on the weird ideas of my own.

Thanks to all my friends for being there, and for listening to me talk about operating systems, memory layouts, heap allocation and garbage collection over and over these last few months. Perhaps especially those of you who don't know the first thing about computers.

Thank you to my family for supporting me during a long and difficult project, for making me relax when I most needed it, and for several supply drops of snacks. Thanks also go to my dad, Ole Kristen Solbjørg, for proofreading my final draft.

I would also like to thank my partner, Martine, for being a great support through the ups and downs of writing the thesis, for brewing countless mugs of coffee — and for reminding me to go for a walk now and then after working long hours and late nights.

Rikke Solbjørg
Trondheim, July 2021

Contents

Figures

Tables

Code Listings

Chapter 1

Introduction

RISC-V is a quickly growing alternative to other instruction set architectures, such as x86 and ARM; it stands out as a completely free ISA. However, there is still much to be explored when it comes to RISC-V, as it is still rather new. In an effort to begin exploring the possibilities of RISC-V, we previously undertook a project to port Project Oberon 2013 to RISC-V [1]. This thesis builds on that project, further exploring the possibilities available in running the Oberon system on RISC-V.

The Oberon system, along with its corresponding programming language and compiler, was first developed by Niklaus Wirth and Jürg Gutknecht between 1986 and 1989 [2]. In Niklaus Wirth's well-known article A Plea for Lean Software, it is given as an example of software that is small, simple, and easy to understand [3], as a potential counter to a version of Parkinson's Law which states that "software expands to fill the available memory". Furthermore, Wirth's newest iteration of the system, Project Oberon 2013, is published as free and open-source software, making it easy to access and research[1].

In this thesis, code overlays will be explored as a way to further minimise the memory footprint of the Oberon system. Overlays became less of a necessity with the increased adoption of virtual memory. A memory management unit (MMU) can offer the illusion of an infinite amount of virtual memory mapped onto a restricted amount of physical memory, supplanting the need for overlaying techniques.

However, requiring an MMU increases the complexity of hardware, which is undesirable for very simple embedded systems. There has recently been increased interest in using overlays on embedded RISC-V systems, and a task group has been created to that end [4]. RISC-V's modularity and extensibility makes it particularly useful for embedded/IoT applications, as a device can be very simple and still be compliant with the specification. With this in mind, a RISC-V implementation with a minimal amount of extensions, use of only machine-mode, and no MMU, is now a hardware platform with real-world application.

[1] The entirety of the Oberon system can be found on Prof. Wirth's website, `https://people.inf.ethz.ch/wirth/ProjectOberon/`.

A website with additional resources can be found at `http://projectoberon.com`.

To our knowledge, implementing code overlays on Project Oberon has not been attempted before. This is despite the fact that Project Oberon is an operating system designed with a single, physical address space in mind, with the goal to be as minimal as possible [2][3]; code overlays would minimise it further. In addition, as mentioned, RISC-V is seeing renewed interest in code overlays in general. Therefore, implementing code overlays on the Oberon system under RISC-V is a goal worth pursuing.

Thus, the overarching research goal of the thesis is to **evaluate whether code overlays are a tenable strategy for reducing the memory footprint of the Oberon system**. For it to be a tenable strategy, it must be able to reduce the memory footprint while exhibiting acceptable performance. To that end, the thesis will explore different strategies for implementing code overlays in the Oberon system, and evaluate whether any of them either fulfil this goal or have the potential to fulfil this goal. In addition, we also hope that in using an operating system that has not been widely researched in quite some time, novel insights will reveal themselves.

Additionally, while it would be possible to manually create overlays based on careful analysis of the operating system, this is a difficult task for several reasons, biggest of all being that it is error-prone in a regular application [5][6], let alone an entire operating system [7]. Furthermore, automatic overlays will allow any program to run on top of the developed overlay system without programmer intervention. Therefore, automatic generation of code overlays will be pursued.

The thesis is laid out as follows. Chapter 1 has introduced the overarching goals for the project. Chapter 2 introduces concepts regarding RISC-V, Oberon, memory management, and code overlays, that will be important to the rest of the thesis. Then, Chapter 3 will cover how the goals presented above were reached, as well as the decision-making process behind certain choices. Chapter 4 covers the ways in which data was gathered, limitations faced regarding what data could be gathered, and the specific experiments that are run. Chapter 5 presents results on the data that was gathered and discusses them. Chapter 6 evaluates the presented strategies, and proposes future avenues to further develop the concepts presented in this thesis, before the thesis is concluded in Chapter 7.

Chapter 2

Background

In the following chapter, background information useful for understanding the rest of the thesis is presented. In Section 2.1, basic information on the RISC-V instruction set architecture — the one used in this thesis — is given. Section 2.2 gives an introduction to Project Oberon, its distinctive characteristics, as well as the process of porting it to the RISC-V instruction set architecture, which was completed in a previous project. Finally, Section 2.3 covers prior work and important details to note on code overlays, both in general and in the context of operating systems in particular.

Note that parts of this background section are adapted from the one given in our previous project report on porting the Oberon system to RISC-V [1], which can be found in Appendix B.

2.1 RISC-V

RISC-V is a rather new instruction set architecture, the goal of which is to become completely universal [8]. In other words, it should be able to accommodate all possible cores that desire to implement it, whether they are in-order or out-of-order; as well as all technologies a core can be fabricated with, whether it's on an FPGA or ASIC.

It is unique for many reasons, and not all of them will be recounted here. Of particular importance to this project, RISC-V is an example of a RISC (reduced instruction-set computing) design, meaning it favours combining multiple simple instructions to do something complex, as opposed to performing the complex instruction in hardware. This is as opposed to a CISC (complex instruction-set computing) design, wherein the processor understands many more instructions that can perform very specific operations.

Another important aspect is that it is a *modular* ISA [8]. It offers a basic set of instructions that every RISC-V processor is guaranteed to implement, and then a set of extensions that a processor can choose to implement depending on what it targets. For instance, a small implementation might use `RV32I`; the RV signifies

that it's RISC-V, *32* signifies a 32-bit processor, and *I* signifies the most basic extension, which includes only instructions essentially deemed necessary, 47 in total. Other extensions can be added on top of this: for instance, instructions for multiplication and division are defined in the *M* extension. An implementation that also includes these instructions would be a `RV32IM` processor. There are many other extensions, such as F and D for single- and double-precision floating point respectively.

Choices regarding what parts of the RISC-V ISA to support were done in our previous project porting the Oberon system to RISC-V[1], detailed in Appendix B. These choices will be reiterated for the convenience of the reader. A `RV32IM` architecture was targeted, as it is a reasonable architecture for embedded applications, including the kinds of constraints under which Oberon performs well. As mentioned, that means basic instructions, as well as multiplication and division instructions. Notably, it also means that floating-point operations are not supported by hardware. Another feature offered by RISC-V is various modes that signify different levels of privilege. It offers three: machine-mode, supervisor-mode, and user-mode [9]. While many modern operating systems require different privilege levels, this is not necessary for the Oberon system. As such, a system using only machine-mode was targeted, often ideal for simple embedded systems [9].

2.2 Project Oberon

This section will go over several aspects of Project Oberon that are useful to have a basic understanding of for the rest of this thesis. Section 2.2.1 covers the history of Oberon, and more importantly the goals it strives to achieve. Section 2.2.2 will explain some basic aspects of the Oberon programming language, which the entire Oberon system is programmed in; Section 2.2.3 will cover specifics regarding the Oberon compiler, which will see several modifications throughout the thesis; and finally, Section 2.2.4 will cover the construction of the Oberon system itself, particularly the aspects relevant to this thesis.

2.2.1 History and goals of Oberon

The Oberon system was first designed in 1986–1989 by Niklaus Wirth and Jürg Gutknecht, then as a complete workstation for use both in academia and industry, on which they also wrote a book describing the inner workings of the entire system [2][3]. The goal of this textbook is to make the system easily understandable to any single person who takes the time to study it. As mentioned in Chapter 1, it was used as an example of a lean software system in [3], where it is cited as a simple, minimal, yet complete system, something that could only be done by focusing on the essentials. It was later picked up again by Wirth in 2013, to update the system and provide an accompanying textbook equivalent to the previous — as the old textbook was based on a machine that is no longer available, the Ceres, using a processor that is also no longer available, the NS32032 [2].

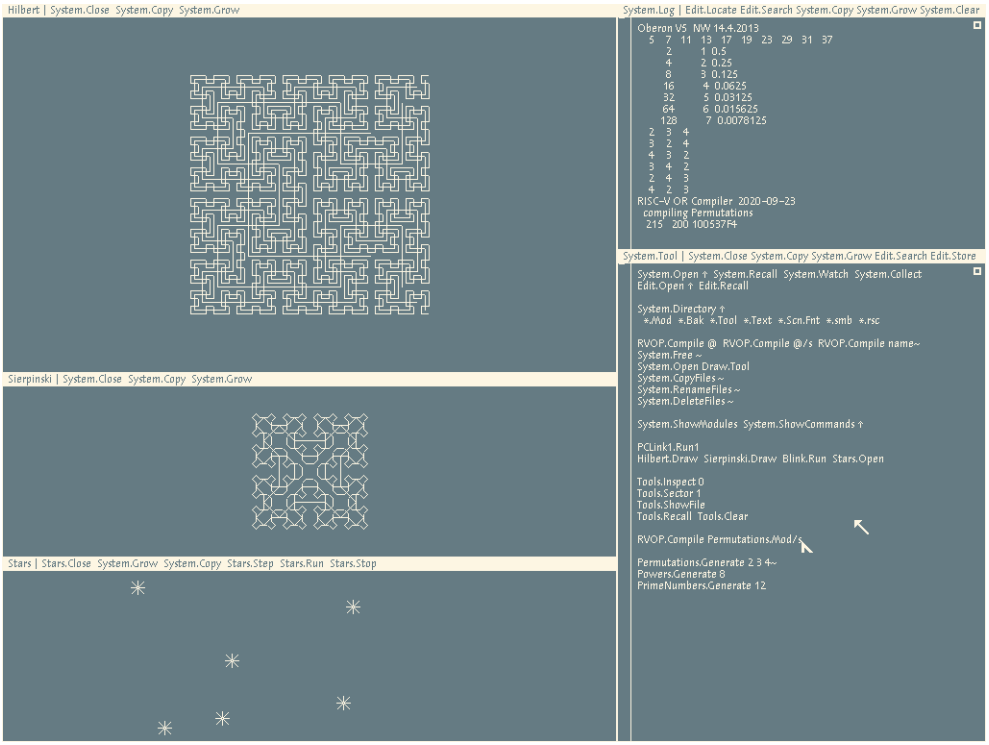


Figure 2.1: A screenshot of the Oberon system (here running on RISC-V). Taken from our previous project performing this port, included in Appendix B.

A particularly notable addition in the 2013 version is the design of a custom RISC processor, to replace the NS32032. Both the processor and the ISA were called RISC-5 (not to be confused with RISC-V), and they were designed with the goal of being very simple, such that the hardware can also be studied and understood.

For the remainder of the thesis, it will be useful to understand the goals of the Oberon system, as presented in [3] and [2]. The primary goal "was to show that software can be developed with a fraction of the memory capacity and processor capacity usually required, without sacrificing flexibility, functionality, or user convenience" [3], while the secondary goal "was to design a system that could be studied and explained in detail, a system suitable as a software-design case study that could be penetrated top-down and whose design decisions could be stated explicitly" [3]. These stated goals remain the guiding principles of Project Oberon 2013 — its largest changes to previous versions are by far found in the compiler, and not in the system itself [2].

Finally, as an example of what the Oberon system looks like, the reader may find Figure 2.1 interesting. This figure in particular is of the ported RISC-V system, but it should look no different from a RISC-5 Oberon system (apart from a differently named compiler).

2.2.2 The Oberon language

The Oberon language is a successor to previous languages in the Wirth family of languages, such as Pascal and Modula. Its design sensibility is quite similar, but it has a more stripped back set of features, with the goal of making it "as simple as possible, but no simpler" [10]. It is meant to be general-purpose, while satisfying enough criteria to be used as a systems programming language. A few features of the programming language, namely those particularly relevant to this thesis, will be explained in this section. The full report of the most recent revision of the language can be found at [10].

Firstly, Oberon programs are split into units that can be separately compiled, i.e. modules. A module is a collection of constants, types, global variables, and procedures [10], and a module can import other modules (so long as the dependency graph does not become circular). Furthermore, as long as a dependent module's exported types/procedures have not changed, recompilation of a module does not necessitate recompilation of dependencies. This is to allow easy extensibility without forcing unnecessary recompilation of every imported module. The programmer chooses which parts of the module to export for use by other modules, e.g. from its types and procedures.

Another important feature of the Oberon language is that it is strongly typed. It offers a set of basic types (such as BOOLEAN, CHAR, INTEGER), the potential to create arrays, pointers, and types describing specific procedure signatures, as well as the potential for the programmer to create new types, known as record types (which are quite similar to structs in C). A consequence of this strong typing is that, unlike a language like C, pointers are also strongly typed: pointers *must* point to a specific record.

However, as it is meant to work as a systems programming language, the Oberon language also specifies the existence of the pseudo-module `SYSTEM`. This module contains procedures that allow for e.g. breaking the rules of the language, as well as procedures that are implementation-dependent, both in terms of the compiler and the underlying computer. An example of a procedure that breaks the rules of the language is `SYSTEM.VAL`, which allows for converting one type to another (i.e. casting). An example of a procedure that is specific to the machine in question, on the RISC-5 system, is `SYSTEM.H`, which reads the auxiliary register H on the RISC-5 processor, containing the upper 32 bits of the previous multiplication operation or the remainder of the previous division operation (depending on which was last performed) [11].

Finally, a word on terminology. Although the term "function" commonly encompasses any callable unit in modern computer science, the Oberon language and documentation makes a consistent distinction between a function, which returns a value, and a procedure, which does not return a value. As such, when discussing particulars written in the Oberon language, this thesis will make the same distinction.

2.2.3 The Oberon compiler

Several characteristics of the Oberon compiler are particularly noteworthy for this project. The characteristics described here were present in the original RISC-5 compiler as written by Niklaus Wirth, and are also in the RISC-V port.

Firstly, the Oberon compiler is a single-pass compiler. Information contained later in the program cannot be used by earlier parts; for instance, a procedure declared later in the program cannot be called by an earlier procedure except by means of an indirect function call. It also performs no optimisations: it contains no abstract syntax tree or intermediary representation, instead emitting machine code on a per-statement basis. This has ramifications for precisely how performant a compiled Oberon program can be. For example, take the following procedure:

Code listing 2.1: Simple example of an addition procedure in Oberon.

```
1 PROCEDURE Add(a, b: INTEGER): INTEGER;
2   VAR c: INTEGER;
3 BEGIN
4   c := a + b;
5   RETURN c
6 END Add;
```

An optimising compiler could turn this procedure into two instructions, adding the two registers containing the values of variables `a, b` together and depositing the result in the register holding the return value, then jumping back to the caller. It could even be inlined, turning it into a single instruction.

The Oberon compiler, however, will scan the program one line at a time, and perform no analysis of the program to cut intermediate steps that are unnecessary. In this example, it will first increase the size of the stack to accommodate the variables, push the parameters onto the stack, pop them from the stack, add them together into a new register, push that register back onto the stack in the location reserved for variable `c`, then pop that off the stack into the register reserved for the return value before jumping back to the caller.

In other words, Oberon programs are often less performant than what one is used to in modern compilers. This is in large part because Wirth's metric for the quality of the compiler was not the performance of all compiled programs, but rather the *speed of self-compilation*, a metric either invented or popularised by Wirth [12]. In other words, the compiler is its own benchmark, and the only improvements allowed to enter the compiler are those that increase the speed at which the compiler compiles itself. This results in a very fast compiler, though at the cost that it does not employ many optimisations.

As this thesis will not focus on compiler optimisations, these design decisions have been left as-is; however, they do have implications for how performant it is possible to make programs that are built using the Oberon compiler.

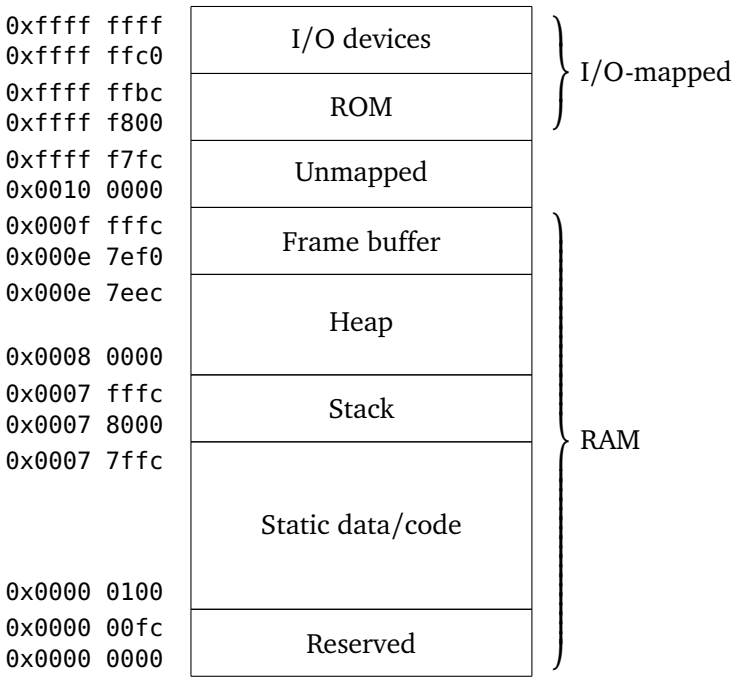


Figure 2.2: Memory layout of the Oberon system.

2.2.4 The Oberon system

The Oberon system has some unique characteristics that will be of particular interest in this thesis, which will be discussed in the following section, giving a detailed description of specific parts of the Oberon system. The memory layout of the Oberon system will be explained in some detail, as understanding it is of particular importance when trying to improve the memory footprint of the entire system. This is followed by an explanation of Oberon's dynamic linking loader, which will see some significant modifications in the prototype presented in this thesis. This is followed by an overview of the process of booting the Oberon system. Then, an interesting feature of the system is briefly explained, namely the built-in heap allocator and its corresponding garbage collector, responsible for deallocation. Finally, a brief explanation of the process of porting the Oberon system to RISC-V is given.

The Oberon memory layout

An understanding of the Oberon memory layout, depicted in Figure 2.2, will be useful, as several modifications to it will be described later in the thesis. The Oberon system's memory layout is split into three sections: static data/code, the stack, and the heap. Additionally, the upper areas of memory are used for the bit-mapped display and I/O.

The "Reserved" area is used for the trap handler as well as the module table. The module table is a table of pointers into the static data/code section, which is the largest region of memory in the system. This is where the linking loader places modules, allocating space for static data, type signatures, and any module code. It effectively functions like a first-fit heap: if a module is removed, it is considered a hole in the list of modules, and upon allocating space for a new module, it looks through the list for any holes large enough, placing it at the end of the list of modules if no appropriate holes are found.

Next, although the stack is given with a definite size here of 32kB (as the system does specify a theoretical "size" of the stack), there is no canary mechanism or similar in the Oberon system to detect a stack overflow, and in practice, few Oberon systems are demanding enough to cause one. (Maliciously breaking the system with a stack overflow is also trivial, but security is given little consideration in the Oberon system in general.)

The heap is the second largest portion of the system, at approximately 425kB. As in other systems, the heap is used for dynamic data allocation.

Finally, the frame buffer section of memory is bit-mapped onto the display. As Oberon only uses monochrome colours, one bit for each pixel suffices. The display is by default assumed to be 1024x768, so it follows that this memory region must occupy at least 98304B. (In the system it occupies 98576B, as there is some unused memory between the heap and the display buffer; this unused memory is not necessary for the system to function.) This part is not considered for improvement in this project; one could trivially lower the required memory for the frame buffer by lowering the resolution, e.g. by lowering it to 800x600 it would only require 60000B. Different hardware to handle displaying pixels in a more memory-efficient manner could also be employed, but again, that would be outside the scope of this thesis.

As the sizes of the section for static data and code, the stack, and the heap are all variable, they can be tweaked to different sizes depending on the needs of the system.

The dynamic linking loader

As supporting code overlays requires loading already linked code from secondary storage to main memory, it will be useful to have an understanding of how Oberon links modules, especially as Oberon does *not* link modules right after compilation (i.e. statically), but rather in runtime. To support increased extensibility and to avoid large statically linked binaries, the Oberon system uses a dynamic linker and loader [2]; similar advantages of dynamic linking are noted in [13]. The dynamic linker only links an Oberon module right before it is to be loaded into the system, using information only available at runtime, such as other modules' positions in the module table, the exact locations of dependencies in memory, etc., to do so. As Wirth notes in [2], to do so requires the linker to be fast, and therefore quite simple.

The linking loader first reads the header of the module to gather information needed for linking, and then reads the unlinked code into memory. The linker works by traversing linked lists stored in instructions in the code, with the tail of each linked list provided in the header of the module, performing fixups — i.e. changing the instruction to correctly reference the location of external dependencies — on every instruction in each linked list. One such tail points to the location of the last instruction in code that needs a fixup, which contains both enough information to correctly identify how to fixup the instruction, as well as a pointer to the next instruction that needs a fixup. When it finishes traversing every list, the module is done being linked, and as the module was already loaded into memory prior to being linked, this doubles as the module being successfully loaded. Note that in no part of this stage is the linked code written back to secondary storage — the code is only linked in memory, and if the module is unloaded and needs to be loaded anew, the linking process will have to be repeated.

Booting the Oberon system

Booting the Oberon system occurs in multiple phases, and will necessarily interact with the code overlay system presented later in the thesis. Therefore, an understanding of how the Oberon system builds its different layers on top of one another in the process of booting will be helpful. A more detailed description can be found in [2], but it will be quickly recounted here.

Booting Oberon occurs in three phases: loading the inner core, then loading the outer core in two steps; which modules are involved in which stage of the booting process can be seen in Figure 2.3. The bootloader, responsible for loading the inner core, is present in the ROM; the processor knows to begin the booting process by reading instructions from it. Once the inner core has been loaded by the bootloader, the bootloader jumps to the entry point for initialising the inner core, transferring execution from the ROM to main memory. Note that, while the inner core is responsible for linking and loading the outer core, it cannot link itself; it has to have been linked beforehand, which is done in the process of installing the operating system on a new workstation.

Once execution has been transferred to the inner core, it loads what has in Figure 2.3 been termed *Phase 1*, which is responsible for loading most of the UI, as well as the driver for the keyboard and mouse. It is also responsible for the module **Oberon**, which contains the core loop that governs the operating system, as well as procedures for calling other modules via user input. Once *Phase 1* has been loaded, it immediately begins loading *Phase 2*, which builds on top of *Phase 1*, before entering the previously mentioned loop. At that point, the system is fully booted.

Heap allocation and garbage collection

The Oberon system has both a heap allocation system as well as a garbage collector built into its kernel. These will both play a role in this thesis, so an introduction

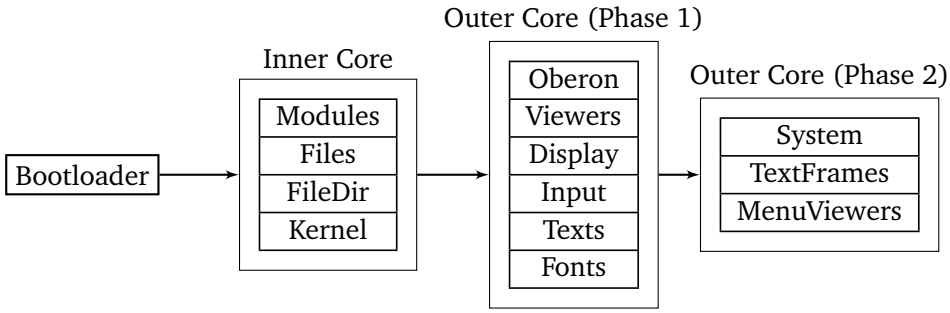


Figure 2.3: The booting process of the Oberon system.

Figure 2.4: The structure of every allocated block on the Oberon system heap, based on a figure in [2].

to how they work is warranted.

The heap, in short, uses free-lists segregated by block size. To avoid fragmentation, the heap is composed of four lists of free blocks at different levels of granularity: 256B, 128B, 64B, and 32B. Allocation of, for instance, a 32B block is done by allocating a 64B block, marking the first 32B as allocated, and marking the other 32B as free in the 32B free-list. The same goes for every granularity, up to the 256B list, which allocates as many 256B blocks as are needed for the size of the data structure being allocated. By the taxonomy of dynamic storage allocators presented in [14], the Oberon heap allocator uses a system of segregated fits using strict size classes with rounding.

To mark whether a block in the heap is free, as well as to denote the size of the block, two words are reserved as a prefix to every block. If the block is allocated, the first word is a pointer to a field in memory that holds the size of the block; if it is freed, the first word is the size of the free block. The second word is used to mark whether the block is allocated in Oberon's mark-scan garbage collection scheme. Thus, for every allocated block, an extra 8B is required, as can be seen in Figure 2.4.

Oberon also includes a built-in garbage collector, which is installed as a task that runs every second (provided the system is not occupied running another process). This task is located in the **Oberon** module, which calls procedures in the kernel to perform the different phases of garbage collection. As Oberon employs a mark-scan (also known as mark-sweep) garbage collector [2], there are two phases, marking and scanning, both of which will be explained below.

The mark phase occurs first, and consists of iterating through a list of *roots*. A root is a known pointer, which may point to a data structure that contains other

pointers. Thus, the task of the mark phase is to iterate through this list of roots, mark the pointers in that list, and also search through the data structures pointed to by these roots for additional pointers that must be marked. Although this phase is implemented as an iterative procedure to avoid stressing the stack [2], the actual algorithm is recursive in nature, as it must mark every found data structure starting from each root using a tree traversal algorithm. As Oberon only runs garbage collection when no other process is running, the list of roots is limited to pointers located in the static data section of every module, of which a separate list is kept for each module, precisely to make the process of traversing this list simple.

The second phase is the scan phase. The scan phase linearly traverses the heap, starting from the base of the heap. It scans the prefix of each block on the heap, retrieves its size from that prefix, and uses that size to jump to the next block. While scanning, it keeps track of the *mark* located in each block's prefix. If the mark is 0, it is *unmarked*, i.e. it was not found in the previous mark phase, meaning it will be be collected by adding it to the free-list and setting its mark to -1. Adjacent unmarked blocks are merged. If the scan phase reaches a marked block, meaning it should not be collected, it is skipped, and its mark is set back to 0 such that it has to be marked again the next time garbage collection runs. If the scan phase reaches a block marked as -1, i.e. free, it is skipped entirely. (Any negative mark will be treated the same way, but the system only uses -1.)

When both phases are complete, the heap has been garbage collected, and the routine is completed.

Porting the Oberon system to RISC-V

The process of porting Oberon to RISC-V is detailed in [1], which is also in this document in Appendix B. A brief explanation of the process follows, as it was essential to allowing this project, and it illustrates several key aspects of the Oberon system.

The porting effort began by setting up an environment in which the Oberon system can both be built and tested. The building system was based on Project Norebo, by Peter de Wachter[1]. Project Norebo is based on a headless version of Oberon that runs on a RISC-5 emulator, with some facilities allowing it to interact with the shell (for printing text) as well as the underlying file system (to read/write files). Using this, Oberon programs such as the compiler can be run; this can then compile the RISC-V compiler, thus setting up a cross compiler running on RISC-5 that emits RISC-V code. The testing was handled by extending an emulator of the Oberon system, also by Peter de Wachter[2], to support RISC-V instead of RISC-5[3].

[1] Project Norebo can be found here: `https://github.com/pdewacht/project-norebo`

[2] The RISC-5 emulator can be found here: `https://github.com/pdewacht/oberon-risc-emu/`

[3] Our extended RISC-V emulator can be found here: `https://github.com/solbjorg/oberon-riscv-emu`

Next, as the built-in Oberon compiler only supported the special-purpose RISC-5 ISA, it needed to be rewritten to support RISC-V. We discovered partway through the process that an in-progress version of such a port existed, courtesy of Samuel A. Falvo II[4], which we built the rest of the project on top of. Though that version was mostly working in that it had been converted to emit RISC-V code, there were a fair few bugs to be ironed out (as is to be expected when testing a compiler). After a considerable amount of effort, the RISC-V compiler reached a state of being fully self-compiling.

Besides porting the compiler, some other parts of the Oberon system necessarily also had to be changed to support RISC-V, though not as many as one would expect in porting an operating system. This is in large part due to the simplicity of the system, as detailed in Section 2.2.1. In particular, the minimal hardware requirements of Oberon made porting it quite effortless once the compiler was fully functioning — as mentioned in Section 2.1, floating-point instructions are not required to run the Oberon system, and in addition, interrupts are not required either. Furthermore, the bootloader of the Oberon system is also written in Oberon, by signalling to the compiler that the module in question should be compiled as bare-metal code, and not as an Oberon module meant to be loaded into the Oberon runtime. Thus, while the compiler had to be changed somewhat to support compiling bare-metal code for RISC-V, the bootloader itself could remain completely untouched. Similarly, the kernel required some minimal changes, while the file system and the outer core of the Oberon system required no changes at all.

One more aspect of the system had to be updated, namely the linking loader. The linking loader, as explained in Section 2.2.4, is responsible for fixing dependencies and loading a module into the Oberon system. Fixing these dependencies proved to be significantly more complex in RISC-V than in RISC-5, due to more complex instruction encodings, though not prohibitively so.

A few things are worth noting about differences between the RISC-5 and RISC-V versions of the Oberon system, however. Firstly, code generated for RISC-V is generally somewhat larger than code generated for RISC-5. It is also in some cases slower and in some cases faster than RISC-5 code, due to different strengths inherent to the two ISAs. Secondly, while the static data sections of the RISC-5 Oberon system are practically restricted to 64kB due to compiler limitations [1], static data sections of the RISC-V Oberon system are restricted to 2kB (due to the load instruction's signed 12-bit offset). In practice this does not have a large effect on the system, except that the sector map in the kernel is stored on the heap instead of static data, and other programs must be rewritten to use the heap if their static data exceeds 2kB. Finally, as mentioned, the linker is more complex.

To summarise, once the compiler, certain aspects of the kernel, and the linking loader were ported, the entire Oberon system worked in just the way one would expect it to. This speaks to a strength of the Oberon system: as all of it is written in the Oberon language, rather than e.g. having parts written in assembly code, it

[4]His RISC-V compiler can be found here: `https://github.com/sam-falvo/project-norebo`

is rather easy to port. It is fully featured, aside from some of the extra hardware features that were dropped (i.e. interrupts and floating-point), and very stable[5].

2.3 Overlays

Code overlays are an old technique in computer programming. By splitting a larger program into smaller independent segments, and *overlaying* them on top of one another, the technique allows a large program to fit into much smaller memory. Overlaying in particular means that two different segments can occupy the same area of memory, depending on when either is needed.

As code overlays are the main focus of this thesis, this section provides some background on code overlays and different ways they are used, as well as relevant related work. Section 2.3.1 will go over the basic structure of code overlays, including the units of which they are composed as well as how they are used in the context of a program. Section 2.3.2 will motivate the use of an automatic approach, rather than a manual one, as well as detail an approach to automatically overlaying an operating system kernel.

2.3.1 An overview of overlays

There are three parts of the technique that immediately must be cleared up: Firstly, what are the smaller segments of the program? Secondly, how are the contents of these segments decided upon? And thirdly, how are these smaller segments overlaid on top of one another? All three of these questions will be given more attention in the rest of the thesis, but an introductory explanation is given here.

Granularity

The first question, what comprises the specific smaller segments that are put together in overlays, is a question of granularity. Different overlay schemes will decide on different levels of granularity depending on the use case in question. For instance, an early example of an automatic overlay generation scheme from IBM [5] uses *modules* as its level of granularity. Modules are here understood as individual files containing several functions. Function-level granularity is also very commonly used [7][15], where an overlay is composed of one or several functions. Another possibility is a sub-function granularity, which splits functions where necessary to improve the memory footprints of large functions [15]. Finally, although not found much in more recent years, another example of a type of granularity was found in the GIER ALGOL compiler from 1965 [16]. An early version of an automatic overlay scheme, it generated what can be thought of as prototypical automatic overlays by linearly segmenting the program into smaller parts comprising a set amount of words. These pieces were automatically swapped into memory; efficient use of this scheme would have to avoid e.g. placing a loop in

[5]Our RISC-V port can be found here: `https://github.com/solbjorg/oberon-riscv`

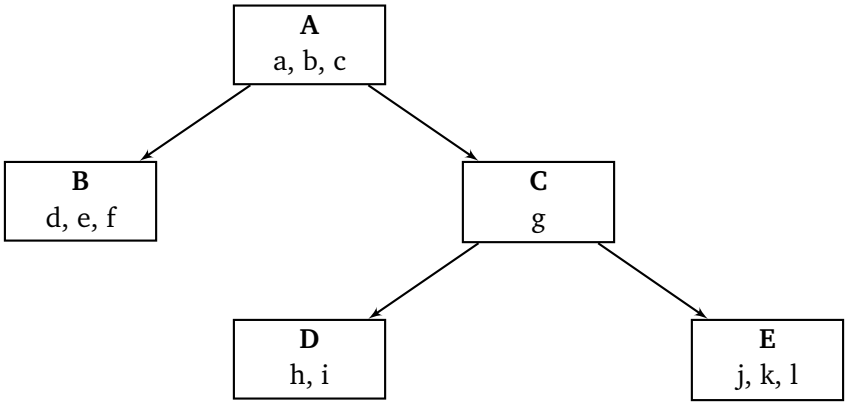


Figure 2.5: An example of an overlay tree.

between two segments, as that would cause it to repeatedly load each segment as the loop iterates [17].

Contents of an overlay

The contents of an overlay are determined both by the granularity of the overlays as well as the specific behaviour of the program. Commonly, it has been the programmer's task to determine which code segments can be overlaid on top of one another, manually creating an overlay tree [13][5]. An example of such an overlay tree can be seen in Figure 2.5. The overlays are named **A**, **B**, **C**, ..., and are composed of code segments *a*, *b*, *c*, ..., of which the segments' sizes determine how many can fit into a single overlay. In this example, overlay **A** is the root of the overlay tree, and must always be resident in memory, such that overlays below it in the tree are able to return to it after finishing execution. Overlays **B** and **C** can be overlaid, as there is no edge between them; the same goes for overlays **D** and **E**. The longest path in this overlay tree is from **A** to **C** to **D** and **E**, where if code in overlay **E** is executing then **A** and **C** must also be resident in memory. If the overlay tree is manually generated, then the programmer must be careful to overlay things in such a way that a tree requiring a minimal amount of memory, while having decent performance, is generated.

A useful way of determining which code segments should be in memory simultaneously is interference, a measure of how many times code segments call one another [6]. If code segments that frequently call one another are placed in the same region of memory, such that the calling code segment will repeatedly have to be replaced with the called code segment (and vice versa for returning), performance will suffer. Interference thus captures an important relationship when considering strategies for code overlays, as a high amount of interference will lead to a large amount of time spent loading code segments into main memory, something that should be minimised as much as possible for acceptable performance.

Overlaying mechanism

After the code segments to be overlaid on top of one another have been determined, a missing piece is still how the program actually loads code segments from disk to memory as they are needed. This is usually done by the overlay manager (OVM), which has the responsibility of intercepting jumps to code that is currently not loaded, and loading the necessary code segments [13][18]. Such interception requires some cooperation from the linker, which has the responsibility of generating overlays and fixing up code such that it cooperates with the overlay manager. A basic overlay manager would work in the following way, where A and B are code segments of any granularity:

1. A needs to jump to a location in B
2. A calls the overlay manager, which checks if B is loaded into memory
    a. If B is loaded into memory, then the overlay manager branches to B, passing along the parameters from A (if there are any)
    b. If B is **not** loaded into memory, then the overlay manager calls a code segment loader that loads B, then branches to B, passing along the parameters from A (if there are any)
3. B executes
4. B returns to A, which must still be in memory

There are, however, a few points worth considering that potentially add to the complexity of an overlay manager.

Firstly, for reasonable performance, item 2*a* needs to be efficient, as every call to overlaid code has to perform this check.

Secondly, in items 2*a* and 2*b*, parameters are passed through the overlay manager to the code segment being called. The exact mechanism for doing this will necessarily be dependent on the calling convention in use. To give a simplified example, if parameters are entirely passed on the stack, the overlay manager can quite easily transparently forward the procedure call by modifying the stack pointer. If parameters are passed entirely using registers, then the overlay manager must ensure that the registers have the correct parameters loaded before calling B. Whatever solution is used will be complicated by the overlay manager having to call other procedures in the case of 2*b*, as it must then make sure to efficiently save and restore parameters as needed.

Thirdly, in item 4, it is necessitated that A is still in memory, as otherwise, if A is not in memory and has been rewritten, it will return to where A branched to B, which may now be data or a different code segment, likely causing the program to crash. This is what gave rise to the tree approach of laying out code segments that can be overlaid. An alternative is to intercept returns as well, and perform a similar series of steps as given above. If this is done, any code segment can be overlaid with any other [7][6].

2.3.2 Automatic overlays

Cytron, et al. mention a few motivations for automating the generation of overlays in [5], and although the paper was written in 1986, these considerations are still relevant today [6]. To summarise, automatically generated overlay structures are desirable because, not only does it save the programmer from having to redesign the overlay tree following small changes in code, but it also greatly reduces the chance of programmer error. Programmer error can mean simply not finding an ideal overlay structure through manual inspection — as they can be quite complex — but also outright mistakes that will lead to a broken program, e.g. if two codependent code segments are overlaid on top of one another (assuming a tree structure as described in Section 2.3.1).

The decision to use automatic overlays was made in a similar project to this thesis, where automatic overlays were used to reduce the memory footprint of the Linux kernel for embedded applications [7]. Their strategy in particular relied on generating overlays after linking through static analysis of the kernel's control flow graph, to find ideal clusters of code. The strategy required a certain amount of kernel code to always be resident in memory, such as trap/interrupt handlers, memory management, etc. All other code could be overlaid, using a separate code buffer that held one code overlay at a time; if a new overlay was needed, the calling overlay was evicted to make room for the new one. This will, in the rest of the thesis, be referred to as a *code buffer* strategy.

To improve performance, certain parts of the kernel that constituted hot code, i.e. code that saw very frequent use, was not overlaid. A separate variable γ is used to determine how much code should be allowed to always stay resident in memory, rather than be overlaid. Their study sees performance improve by an order of magnitude by placing hot code in resident memory, as it no longer has to be loaded into memory every time it is needed [7].

Another thing of note is that some particular difficulties with overlaying operating systems (and kernels) are given. However, most of them do not apply to the Oberon system: for instance, handwritten assembly code is given as an example of code that is difficult to analyse, but that does not exist in the Oberon system. Similarly, Oberon's lack of interrupts (at least in the RISC-V version [1]) simplifies a great deal, as the situation wherein overlaid code is interrupted does not have to be considered.

2.4 Multi-objective optimisation and Pareto optimality

In evaluating different strategies of overlaying code, there are many different objectives one can optimise for. Examples of these are worst-case execution time (WCET), average-case execution time (ACET), minimising interference, memory footprint, and code size. Furthermore, many such objectives directly conflict, such that optimising for one will worsen the other, such as code size and performance; loop unrolling, for instance, improves performance at the expense of increased

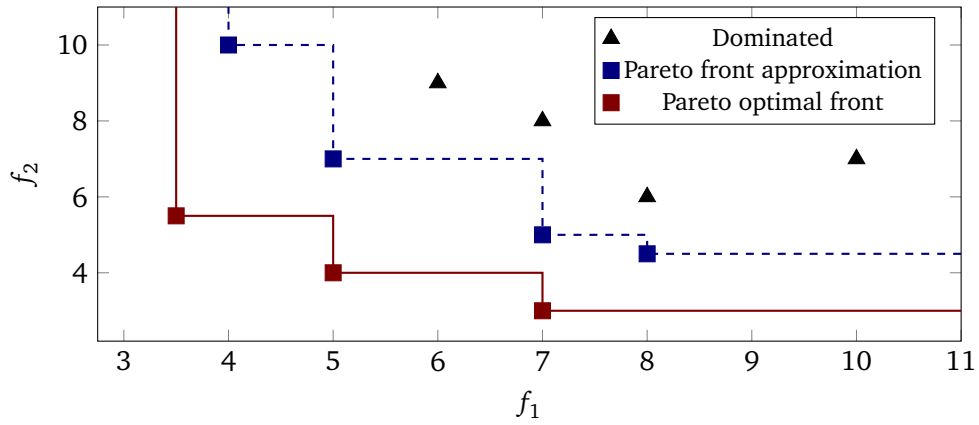


Figure 2.6: The relationship between the Pareto optimal front, the approximate Pareto optimal front, and dominated solutions. Adapted from a figure in [19].

code size. In comparing different overlay strategies, a basic understanding of how one can evaluate their success at optimising for a combination of opposed objectives will be useful. The most relevant objectives to this thesis will be performance and memory footprint.

In [19], multi-objective optimisation is done in the context of real-time embedded systems, within which there are many objectives to optimise for. A similar approach will be used to evaluate different strategies for code overlays, based on identifying approximate Pareto efficient solutions, as well as identifying solutions that are dominated by other solutions. As such, a quick explanation of Pareto dominance, Pareto optimal solutions, the Pareto front, and the approximate Pareto front will be given.

Let a vector in this context be composed of every objective towards which one can optimise; the optimisation in this case will assume that objectives should be as minimised as possible. (If they are to be maximised, multiplying the objective by -1 suffices [19].) A vector u dominates vector v if u is partially less than v, i.e. at least one of its objectives are smaller without any other objectives being higher [20]. A solution is *Pareto optimal* if no other solution dominates it [20].

However, proving Pareto optimality is computationally infeasible for many problems, and as such the goal in this thesis is instead to identify solutions that are approximately Pareto optimal [19]. While we cannot prove whether a given solution is Pareto optimal, we *can* assess whether other solutions are dominated by it. In so doing, a set of potentially desirable solutions are given (depending on which objective one wishes to optimise for), while undesirable solutions are discarded. An example scenario showing the relationship between the Pareto optimal front, the approximate Pareto optimal front, and dominated solutions for two objective functions is depicted in Figure 2.6; squares indicate (approximate) Pareto optimal solutions, and triangles indicate dominated solutions. Finally, note that even if a solution is Pareto optimal, it might have unacceptable characterist-

ics: a solution that optimises entirely for one objective function at the expense of all others may be Pareto optimal, without being desirable; ideally, a good solution should be within acceptable bounds for all objective functions [19].

Chapter 3

Design

In order to support code overlays in the Oberon system, many separate pieces had to be designed and developed.

This chapter begins with an overview of the designed system in Section 3.1. The rest of the chapter is split into two parts. The first part covers the *mechanism* supporting overlays, including compiler support; loading, linking, and creating overlays; and the overlay manager. The second part concerns the different *strategies* for what constitutes an overlay, as well as when an overlay should be loaded into and out of memory. This part also explains aspects that needed to be developed to support the given strategies.

The first part, concerning the mechanism, proceeds as follows. First of all, the overlay mechanism itself, which is responsible for branching from one overlay to another, and loading the new overlay if necessary, is presented in Section 3.2; the process of linking and generating overlays is presented in Section 3.3. Ways to efficiently load code from secondary storage to memory also had to be developed; these are presented in Section 3.4. Finally, a strategy for position-independent code, which avoids every procedure call having to go through the overlay manager, is developed in Section 3.5.

This is followed by an explanation of the different strategies for creating overlays and placing them in memory, and the development thereof. Section 3.6 presents the two strategies by which overlays are placed in memory, one which allows only one overlay to be loaded at a time and another which allows multiple; to support the second method, changes were made to the Oberon garbage collector, presented in Section 3.7. Finally, concrete strategies for creating overlays and placing them are presented in Sections 3.8 and 3.9, which detail module-granularity and function-granularity overlays respectively.

3.1 A design overview

This chapter will detail the design and implementation of the many disparate parts that, put together, make up the overlay system developed in this thesis. To aid the

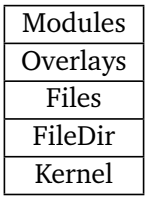


Figure 3.1: The modified inner core of the Oberon system, when code overlay support is added.

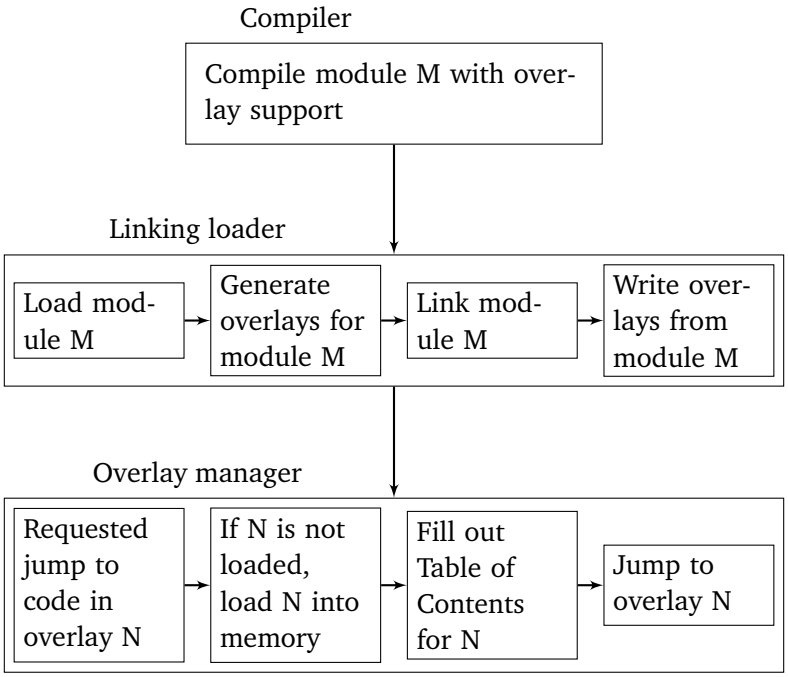


Figure 3.2: The overarching design that will be detailed in Chapter 3.

reader's understanding, this section will give an overview of what each part of the system is doing, and how they fit together.

As noted in Section 2.2.4, booting the Oberon system consists of several phases, starting with the *inner core*, which is responsible for loading the *outer core*. This inner core consists of the kernel, the file system, and the dynamic linking loader. We will be overlaying the outer core, as well as anything loaded on top of it; therefore, the inner core must be changed to support this. This is done by adding a new module, **Overlays**, which is responsible for both the creation and the management of overlays. The new layout of the inner core can be seen in Figure 3.1. As shown, it is placed just below **Modules**, which contains the linking loader. This is because the linker must use the overlay system to correctly link code.

Figure 3.2 shows the overarching design of the system that will be explained in detail in this chapter. It consists of three parts, namely the compiler, the linking

loader, and the overlay system. The overlay system itself is the most significant addition.

Firstly, several changes were made to the compiler, to add information in the header regarding the locations of procedures as well as to correctly handle jumps to overlaid code segments. Additional changes were also made to support position-independent code, such that code can be placed anywhere in memory. Secondly, changes had to be made to the dynamic linking loader. As it is dynamic, the creation of overlays cannot be done before runtime without changing the fundamental design of the linker, as overlays have to be made out of linked code. Thirdly, the overlay system itself had to be built. This is responsible for loading, evicting, and branching to overlaid code. All these aspects will be briefly described here, before they are given a more detailed explanation in their respective sections.

We will in this section first pay attention to the *mechanism* by which overlays are created, loaded, and branched to, followed by an overview of the *strategy* that governs what constitutes an overlay, and when they should be loaded into and out of memory. These explanations are purposefully left somewhat superficial; the details of each part are given in the rest of this chapter.

3.1.1 Mechanism

The overlay manager itself is the part of the system responsible for loading and branching to overlaid code. It is called when a program branches to an overlaid procedure that currently is not in memory, as well as to resolve an indirect procedure call to an overlaid procedure. When it is called, it resolves which overlay needs to be branched to, loads it into memory if needed, and then branches to it. This step also includes filling out a data structure known as the Table of Contents, a table containing the addresses of procedures currently loaded into memory. By using this data structure, programs can branch directly to overlaid code if it is currently in memory, thus both making the code position-independent while avoiding that all procedure calls to overlaid code must be resolved by the overlay manager. Additionally, to improve the performance of loading overlays into memory, we developed a mechanism to read a certain amount of data directly from secondary storage to a given location in memory, rather than to do this through the Oberon file system.

As shown in Figure 3.2, the linking loader now has the additional task of generating overlays for a given module and writing those overlays to secondary storage after they have been linked. These tasks are interleaved, as linking now requires information on the layout of the overlay table. To briefly explain the process: overlays are placed in the overlay table, then the module is linked using these entries in the overlay table, before finally writing the code covered by these overlays into secondary storage.

To support calling the overlay manager when needed, and to provide enough data about a module for the linking loader to correctly link it with overlay support, several changes had to be made to the compiler. As these consist of many small

but necessary changes, they are best left explained to the sections where changes to the compiler were required.

3.1.2 Strategy

The development of strategies for the overlay system happened in stages, where each one builds on the previous. Three major strategies were developed in total; a brief description of each is given below.

Module-granularity overlays using a code buffer

The first, and simplest strategy, is one working on module-granularity overlays. This strategy creates a distinct overlay for every module, and can keep one overlay at a time in memory, using a code buffer. Whenever a procedure call from one overlaid module to another occurs, the code branches to the overlay manager. The overlay manager loads the requested module (overwriting the previous one), fills out the Table of Contents with procedures in the loaded module, and then branches to the requested procedure within that module. When the code returns to the calling module, the called module is evicted and the calling module is loaded back into memory.

Function-granularity overlays using a code buffer

The second strategy changes the granularity of the overlays. In this strategy, each overlay is instead made from a single procedure. The broad strokes of this strategy are otherwise the same as the previous one, meaning only one overlay can be present in the code buffer at a time.

Function-granularity overlays using the system heap

The final strategy builds on the previous one, but changes the method by which overlays are loaded into and evicted out of memory. In this strategy, one overlay is still created from a single procedure, but multiple overlays can be present in memory simultaneously. This is done by using the system heap, which previously was only used for dynamic allocation of data; the heap allocator was changed to also support allocating space for code.

Furthermore, to support evicting overlays from the system heap, the garbage collector was modified to collect overlays that were no longer needed. In addition, due to the design of the Oberon garbage collector, it could only run when no other programs were running; therefore, it also had to be changed to be able to run at any time during program execution.

The part of the strategy determining which overlays should currently be in memory and which overlays should be evicted is performed by communicating with the garbage collector: overlays that should be evicted are marked as garbage.

The particular strategy implemented in this thesis is one that automatically creates an overlay tree on the heap. Procedures currently in the call tree are kept in memory, while any procedure currently not in the call tree is garbage and will be evicted if more heap space is needed.

3.2 The overlay mechanism

This section is split into three. First, the approach of communicating with the overlay manager is explained, by the use of tokens; then, the overlay manager itself is presented, along with how it was improved for both performance and stack size; and finally, the approach taken to handle indirect procedure calls will be detailed.

3.2.1 Overlay tokens

An overlay token is used to pass to the overlay manager whatever information it requires to perform its tasks. For this purpose, register **x30** was reserved, similarly to the current RISC-V software overlay standard proposal draft [21]. (The choice of register is quite arbitrary, and has no impact on performance.) The compiler emits code to load an overlay token into the predetermined register prior to procedure calls.

The format of the overlay token is determined by the information required by the overlay manager to load the correct overlay and jump to the correct position in the overlay. The most basic overlay token consists entirely of the index into the overlay table containing the relevant entry that needs to be loaded. Additional information will largely depend on the strategy in use. If the overlay only consists of a single procedure, then simply jumping to the start of the overlay is correct, and no more information is needed. However, if it consists of e.g. a module, then the specific procedure being jumped to would also need to be communicated.

3.2.2 Overlay manager

A particular difficulty of creating an efficient overlay manager is Oberon's calling convention. While the calling convention developed for ELF binaries targeting RISC-V uses eight registers for arguments and puts the rest on the stack [22], Oberon's calling convention only uses registers. That is to say, if a procedure has 16 32-bit integer arguments, then each of those arguments will be placed in a register starting from x8[1] in ascending order. This has a few ramifications. Firstly, a specific limitation on the number of allowed arguments must be made, as the number of available registers is finite. Secondly, it has implications for how the overlay

[1]The original RISC-5 compiler starts from R0, which is analogous to x8 in RISC-V. While registers x5-x7 are technically free to be used for this, it would make no difference for the efficiency of the compiled program, while both complicating the compiler and reducing the efficacy of RISC-V compressed instructions if that is added in the future [8].

manager is to be implemented, as it must handle those registers in a way that avoids taking up much space on the stack while simultaneously being efficient.

To address the first ramification, a limit of 12 arguments[2] for all procedures was implemented into the compiler as an assertion. This was chosen as a practical compromise: only one procedure in Oberon used as many as 12 arguments in the first place, and if a procedure truly requires more than 12 arguments, it is not difficult to work around such a requirement., e.g. by placing them in a record on the stack. Therefore, it is not an unreasonable limitation.

The second ramification requires more work. A simplified, early version of the overlay manager looked as follows:

Code listing 3.1: Initial implementation of an overlay manager.

```
1  TYPE Func = PROCEDURE(a,b,c,d,e,f,g,h,i,j,l,m: INTEGER): INTEGER;
2  (* ... *)
3  PROCEDURE OverlayManager*(a,b,c,d,e,f,g,h,i,j,l,m: INTEGER): INTEGER;
4    VAR k: INTEGER; ov: Overlay;
5  BEGIN
6    (* get index into the overlay table from overlay token *)
7    ov := SYSTEM.VAL(Overlay, overlayTableRoot +
8      (SYSTEM.REG(30) DIV 2 MOD 200H) * overlayDescSize);
9
10   (* if overlay is not loaded into memory, call overlay loader *)
11   IF ov.mapped = 0 THEN LoadOverlay(ov); END;
12
13   body := SYSTEM.VAL(Func, ov.mapped);
14   k := body(a,b,c,d,e,f,g,h,i,j,l,m);
15   RETURN k
16 END OverlayManager;
```

(Note that this ignores how to handle the case where the caller overlay has been evicted for the moment.) This procedure works in a very simple manner: all procedures pass in at most 12 arguments. Although these arguments can be of any type, it is irrelevant to the overlay manager what these types are; it only needs to preserve the arguments so that they can be passed through to the called function, which knows what to do with them. Similarly, although not every procedure called through the overlay manager in this way will have a return value, it causes no issues to assume they do. In the Oberon calling convention, return values are placed in the lowest register on the stack (x8). Hence, if the procedure being returned to via the overlay manager does expect a return value, it will be in x8 as the procedure expects; if not, no adverse effects occur, except for a few wasted cycles in the overlay manager loading an unneeded return value.

However, this solution has some flaws that make it untenable. Firstly, it is much slower than it needs to be, due to having to push all twelve parameters on and off the stack repeatedly (as explained in Section 2.2.3). In addition, as every parameter is stored on the stack, every call of the overlay manager requires far more space on the stack than necessary: 48B for the parameters, 4B for the

[2] In reality, the limitation is that only 12 *registers* may be used. The two are equivalent in all cases except for passing arrays, which passes both the array's length and the address of its first element, using two registers as a result.

local variables, and 4B for the return value. If the call stack is e.g. 16 calls deep, and they all go via the overlay manager (which is not unlikely e.g. in the case of mutual recursion), the overlay manager demands an additional 896B of space on the stack. This is also at the cost of performance, as pushing every parameter on the stack is costly. That is a rather hefty overhead cost for calling a procedure. While the cost in terms of stack size could potentially be improved by use of a jump to a subroutine that executes a specific function prologue storing only the necessary amount of arguments (given that the token is changed such that this is known), this would also have a negative impact on performance, as noted in [23], and therefore not be an ideal optimisation.

A scheme that remedies most of these issues is as follows:

Code listing 3.2: Optimised version of the overlay manager.

```
1  PROCEDURE OverlayManager*;
2    VAR ov: Overlay;
3  BEGIN
4    SYSTEM.REGSTACK(20);
5    (* get index into the list of overlay pointers from overlay token *)
6    ov := SYSTEM.VAL(Overlay, overlayTableRoot +
7      (SYSTEM.REG(30) DIV 2 MOD 200H) * overlayDescSize);
8
9    (* if overlay is not loaded into memory, call overlay loader *)
10   IF ov.mapped = 0 THEN LoadOverlay(ov); END;
11
12   SYSTEM.JUMP(ov.mapped);
13   SYSTEM.REGSTACK(9);
14 END OverlayManager;
```

This uses two distinct new `SYSTEM` procedures implemented into the compiler, `SYSTEM.REGSTACK` (short for register stack) and `SYSTEM.JUMP`. `SYSTEM.REGSTACK` tells the compiler to place the base of the register stack at the indicated register for the remainder of the procedure, and is used to preserve argument registers (in line 4) and to preserve the return value (in line 13). `SYSTEM.JUMP` emits a jump to the value in the passed variable without any further preamble (such as saving registers on the stack, etc.). In this way, registers x8-x19, which potentially hold the arguments of the caller, are not touched by the overlay manager until the branch to the overlay's code segment occurs. Finally, the register stack is modified again after the code segment returns, to ensure that a potential return value (stored in x8) is not overwritten by code for returning to previous overlay. These optimisations serve two purposes: firstly, the stack is much less stressed, only requiring 8 additional bytes per call[3], and secondly, as the overlay manager no longer has to push every parameter onto the stack, its performance is much more acceptable. It is quite difficult to improve performance beyond this point, as Oberon's compiler does not optimise (see Section 2.2.3).

However, the program cannot preserve its registers in the same way if the

[3]It would also be possible to move the loaded overlay into a global variable, to further save another word from being saved on the stack. However, this would come at some cost to performance, and keeping a stack of loaded overlays is necessary for most strategies.

overlay loader has to be called. In this case, the program temporarily extends the stack to contain all 12 arguments, calls the procedure, before the arguments are popped off the stack after the overlay loader returns. They only temporarily occupy the stack, and the overlay loader does not make heavy use of the stack, so it is a much lesser issue here than in the OVM detailed in Code listing 3.1.

As a final note, while heavy optimisation at the cost of readability is generally unwelcome in the Oberon system (see Section 2.2.1), it was deemed necessary here, as the overlay manager will see extremely heavy use by the overlaid Oberon system.

3.2.3 Indirect procedure calls

Oberon, similarly to many other programming languages, supports indirect procedure call. However, indirect procedure calls rely on absolute addressing to work: in an unmodified Oberon system, they are implemented by loading the address held in a variable into a register and jumping to the absolute address to which that value points. This assumption does not hold in an overlaid system, where overlaid procedures' addresses may either not be in memory at all, or have been moved to a different location in memory since its address was placed in the variable.

Therefore, some solution that allows indirect procedure calls to overlaid procedures to work is required. The solution chosen was to use overlay tokens instead of absolute addresses for overlaid procedures. The overlay token uniquely identifies which overlay to load if it is not already loaded, as well as where within the overlay to jump, and as such suffices as a replacement of an absolute address. Furthermore, as indirect procedure calls cannot go through the Table of Contents (which will be explained in Section 3.5), a separate mechanism for them to call the overlay manager was required; register `x31` was reserved for this purpose.

However, indirect jumps to absolute addresses must also still be supported. Thus, a way to identify whether a given variable is an absolute address or an overlay token is required. A simple observation allows use of the least significant bit (LSB) of the address for identifying whether it is a token or an address. As a memory address will always be word-aligned[4], its LSB will always be 0; thus, by setting the LSB of the token to 1, that can be used to check whether it is an absolute address or not. This observation has previously been used by [21] and [6].

Upon making an indirect procedure call, the compiler emits the code seen in Code listing 3.3. In this example, `x8` holds the address/token of the indirect procedure call.

Code listing 3.3: RISC-V assembly for handling an indirect function call. Labels have been added for readability.

```
1 andi x9, x8, 1            ; deposit LSB of x8 in x9
2 ; jump over following code if LSB of x8 == 0, as
```

[4]Compressed instructions in RISC-V are also always aligned to the half-word, such that their least significant bit is still 0.

```
3 ; if LSB of x8 is 0, then it is a memory address
4 beq x9, 0, NormalJump
5 addi x30, x8, 0            ; put overlay token in x30
6 jalr ra, x31, 0            ; jump to overlay manager
7 jal x0, Continue           ; branch past below jump to normal address
8 NormalJump: jalr ra, x8, 0 ; jump to normal address
9 Continue:                  ; the rest of the program
```

As the comments explain, the LSB is used to check if it is an absolute address or an overlay token. If the LSB is even, it jumps to where the absolute address is pointing; if not, it loads the token into the overlay token register, and jumps to the overlay manager.

This code is emitted in the program rather than handled by the overlay manager. This was for two reasons: firstly, as mentioned in Section 2.2.3, the compiler does not optimise, and therefore would emit less efficient code for handling this case if it were written in Oberon — as the overlay manager is — and as such any additional functionality added to it would be detrimental to its performance. And secondly, due to the implementation of position-independent code as described in Section 3.5, the overlay manager rarely has to handle cases where the passed token is not an overlay token, as jumps to code that is always resident in memory will bypass it; thus, emitting this code in the program guarantees that the overlay manager will receive a token, allowing a more efficient implementation.

3.3 Linking code and generating overlays

In this section, the changes made to the linker, including how generation of overlays were incorporated into it, will be explained. It will begin with an explanation of how overlays are generated, then an explanation of how this is incorporated into the linker, both for module- and function-level granularity (which will be explained in more detail in Sections 3.8 and 3.9 respectively).

Overlays are generated in a separate **Overlays** module, which contains both the overlay manager and procedures for generating overlays. As the linker requires information on overlays in a module to successfully link it — as it must fixup instructions to load overlay tokens — generating an overlay is done in two steps. First, their entries in the overlay table are generated; then, after the module is linked, the overlays are written onto the disk. In this way, linking of the overlay occurs between when its entry into the overlay table is generated and when it is written to secondary storage. As for the linker, in module-granularity overlays, it could largely function as before. As the linker is located in the inner core, it is not overlaid; therefore, instead of loading the module into the static data section, it is loaded into the overlay region. Hence, the linker will load the module in question into the overlay region, tell the overlay system to generate an entry for it in the overlay table, link it as usual, then tell the overlay system to write it to secondary storage. The entry, which does not have an overlay generated for it, is treated slightly differently, instead being placed at the end of the static data region. This is due to the fact that, if the entry code of a module were to call an overlaid

procedure (as it often does), that overlay would replace the entry code, with no way to recover it. To avoid wasting space in the static data region on unnecessary entry code, the next module's data and entry code is loaded on top of the previous entry code, i.e. it is overwritten.

However, some additional complexity in the linker is required to support linking procedure-granularity overlays in the Oberon runtime, as there is no space in memory to load the entire module — particularly for large modules such as **TextFrames** (occupying 31228B in code). Thus, an alternative solution is required.

One potential solution is to, rather than load the entire code segment into memory, simply keep two files: one for the unlinked version and one for the linked version of the file. Then, fixups could be done by moving around in the unlinked file, reading only one word, performing the fixup on it, and writing that into the linked version. Finally, each procedure in the linked version can be written to disk. However, as will be explained in Section 3.4, the file system in Oberon is quite slow; to use a dynamic linker in the system, it necessarily needs to be performant, which this solution would not satisfy.

A more satisfactory solution, instead, is to link one procedure at a time. One procedure is read at a time from the file into a region in memory, linked, then written to disk. This process repeats for every procedure, until an overlay has been generated for every procedure without having to load the entire module into memory. As detailed in Section 3.6, for a function-granularity strategy using a code buffer, the code buffer can be used for this; for heap-based strategies, a block on the heap is allocated. The solution for handling entry code is carried over from the module-granularity linker.

Something worth noting about this system is that, as explained in Section 2.2.4, the Oberon system uses a dynamic linking loader. In other words, it does not statically link binaries right after compilation, but rather in the Oberon runtime. This design decision was kept, as creating a static linker for the Oberon system is outside the scope of this thesis. However, as overlays have to be created from linked code, this means that, as opposed to most overlay systems that generate overlays beforehand, the Oberon system must generate them after they have been dynamically linked, i.e. in the Oberon runtime.

3.4 Improving performance of transfer from memory to disk

In an overlay system, there is a single very clear performance bottleneck: the transfer of code segments from the disk to memory. Regardless of strategy, this must be done as efficiently as possible, as the critical path of the overlay manager will always involve loading code segments.

One alternative was to use direct memory access (DMA) to transfer large overlays, such as in [6], as this would allow very fast, hardware-assisted transfer of code from secondary storage to memory. However, as a stated goal of the Oberon

system is simplicity both in hardware and software (see Section 2.2.1), adding a DMA engine is not an ideal solution; ideally the overlay system should be performant without one. Therefore, an effort was made to improve the system of loading linearly from disk to memory within the Oberon system itself, without adding a hardware requirement.

Code listing 3.4: Code for copying an overlay from disk to memory using the file system.

```
OverlayDesc* = RECORD
  filename: FileName;
  mapped*: INTEGER; (*address: start of mapped overlay*)
  size: INTEGER; (* length of code segment; excludes footer *)
END;

(* ... *)
(* the routine for transferring from the file system to memory: *)
F := Files.Old(ov.filename);
IF F # NIL THEN
  ov.mapped := regions[0]; (* Map overlay to overlay region in memory *)
  Files.Set(R, F, 0); (* Set R to read from start of file *)
  FOR i := 0 TO ov.size-4 DO
    Files.ReadInt(R, u); SYSTEM.PUT(ov.mapped + i, u);
    INC(i, 4);
  END;
END;
```

The first iteration of this component used the most naive approach, i.e. integrating it with the file system, which can be seen in Code listing 3.4. Although functional, this approach is incredibly inefficient for multiple reasons: it requires interacting with the file system to allocate a buffer from which to read the file; it requires reading from the file word by word, which in the internals of the file system is actually translated to reading byte for byte; it requires first reading the file from disk onto a buffer, and then reading from the buffer into the desired location in memory; and it requires storing the entire filename in the Overlay record, which occupies 32 bytes, thus incurring significant overhead. It is worth pointing out that the file system is meant to be much more general-purpose than what an overlay system requires: as an example, it will conservatively allocate buffers as they are needed to read the file, whereas for the overlay system the entirety of the file will always need to be read.

Therefore, a better solution was necessary. This solution, then, would have to fulfil several conditions to be well-suited to overlays:

- Low storage overhead: It should only be necessary to store the sectors the file is stored in, not the full filename.
- High performance: Since this will see a lot of use, it will necessarily have to be as performant as possible.
- Concise: As the overlay system will always be in memory, this solution should require as little code as possible, to avoid unnecessary use of memory.
- Lack of indirection: It should read directly from disk to memory, without a layer of indirection (such as the Oberon file system's buffers) in between.

Removal of unnecessary copying between buffers in memory has positive effects on both performance and memory footprint [24].

It should be noted that efficient use of disk space is not a stated requirement. As secondary storage is cheap, and we are only interested in optimising for efficient use of memory, it will be ignored in this case.

Although a version that still incorporated the file system was attempted, it violated the first two conditions, while barely achieving the third. As there was no case already in the Oberon system where loading a file directly to a location in memory quickly was necessary or even beneficial, and using the file system proved too unwieldy, a solution had to be built from the ground up.

First of all, this required repurposing the SD card driver in the Kernel module. As the kernel procedure GetSector already read an entire sector from a location on disk to a location in memory, only a few changes needed to be made. The most important change necessary was to stop reading to memory after already having read the entirety of the file, to avoid retrieving garbage. The exported function, for use in the overlay manager, is as follows:

Code listing 3.5: Code for efficiently getting data from disk to memory.

```
1  PROCEDURE GetSectorToMem*(src, dst, lim: INTEGER);
2  BEGIN
3    (* All sector addresses are multiplied by 29 in Oberon, as a simple *)
4    (* redundancy check. *)
5    ASSERT(src MOD 29 = 0); src := src DIV 29;
6    src := src * 2 + FSoffset;
7    IF lim <= 508 THEN
8      ReadSDLim(src, dst, lim);
9    ELSIF lim <= 1020 THEN
10     ReadSDLim(src, dst, 508); ReadSDLim(src+1, dst+512, lim-512);
11   ELSE
12     ReadSDLim(src, dst, 508); ReadSDLim(src+1, dst+512, 508);
13   END;
14 END GetSectorToMem;
```

Note that sector addresses in Oberon are given in 512B granularity, as that is the lowest block size allowed on an SD card [25]. Additionally, the assumed sector size here is 1024B[5]. The mechanism is quite simple: `ReadSDLim` has been written as an extension of the SD card driver to stop reading into memory once a certain limit `lim` is reached (while still retrieving the entire block, as required by the SD card specification [25]), and `GetSectorToMem` uses this to write the necessary amount to memory. An exactly equivalent procedure was written for writing sections of memory to a specified sector on the disk, to be used when generating overlays.

Second, the overlay manager needed to support using disk sectors directly rather than the built-in file system. This was done by replacing the `filename` field in the overlay table with a `sector` field, containing the address of the first sector of the overlay's code segment. This has the advantage of reducing a 32-byte overhead to 4 bytes, as disk sectors use 32-bit addressing. To determine *which* sectors

[5]This assumption also exists in the RISC-5 Oberon system, and must be changed if it does not conform to the SD card used.

to write the overlay to, a simple strategy was used: determine a starting sector to write overlays to, and write linearly through the disk from that point. While previously the file system occupied the entire disk, with this strategy a portion of the disk is allocated exclusively to the overlay system. (As the file system in Oberon somewhat slows down as the number of files in a system increases, this also benefits performance, as an overlay system using the file system would require many files.)

To know how many sectors to read, the overlay's size is used, a field already required for other parts of the overlay system. The logic to perform the transfer from disk to memory can be seen in listing 3.6.

Code listing 3.6: Oberon code for moving a code segment from secondary storage to memory.

```
1 FOR i := 0 TO ov.size BY FileDir.SectorSize DO
2   Kernel.GetSectorToMem((ov.sector + i DIV FileDir.SectorSize) * 29, ov.mapped + i,
3     ov.size - i);
4 END;
```

Note that ov is an instance of the Overlay record. This reads sector by sector, starting from the first sector the overlay is stored in, and reading it into the area of memory allocated for the overlay. (How ov.mapped, i.e. the address in memory to write to, is set, will be detailed in Section 3.6.2.)

3.5 Position-independent code

For code overlays to be feasible, code loaded into the overlay region must be position-independent. Position-independent code invariably requires some additional level of indirection, to resolve any code that would previously use absolute addressing [13]. While there are many different approaches to achieve this, two strategies were chosen: the first strategy was to implement a *Table of Contents*, as used in IBM AIX [13], which is essentially a redirection table. This allows calls to other overlays in memory to be more direct, without having to be redirected by the overlay manager. The second strategy was a hybrid with the previous method, where every call to an overlaid procedure goes through the overlay manager, while calls to memory-resident procedures still go through the Table of Contents.

3.5.1 Table of Contents

To explain why Table of Contents in particular was chosen, it will first be useful to explain how Oberon treats global symbols.

In Oberon, each module has its own data segment. To access a module's data segment, the global data pointer register (from here on referred to as **gp**) first needs to be set to the address pointing to the base of the data segment. Variables are then accessed according to an offset into the data segment. While this offset is known at compile-time, the address of the data segment is not, so the linker will have to perform fixups of all loads into **gp**. This is very similar to how e.g. many

C compilers will treat global and static variables. A major difference, however, is in Oberon's concept of modules, which is absent in C. To load an exported global variable in another module, the value of **gp** must be changed to point to that module's data segment. In short, there is a layer of indirection to loading global variables.

The Table of Contents adds such a layer of indirection for procedure calls as well. In short, it is a table containing the current locations of global variables and procedures. Using a Table of Contents, whenever a global variable or a procedure is used, the program consults this table, retrieves the current location of the variable/procedure, and then loads from the given location [26]. In this implementation, as data overlays are not considered, the table still contains the values of global variables rather than their addresses. This adds some overhead to procedure calls, which now always require consulting the Table of Contents to resolve the procedure's current address. Note that, to support position independence within a module, calls to non-exported functions also require consulting the Table of Contents. In addition, the Table of Contents must remain in memory, occupying a word for every procedure in a module. If a module has e.g. 10 procedures, the memory footprint of its Table of Contents would be 10 words, i.e. 40B. In the Oberon system ported to RISC-V, there is a total of 293 instances of the PROCEDURE keyword, resulting in 1172B of memory expended on the Table of Contents[6].

While the Oberon compiler already added a field detailing the space required for every global variable to every binary's header, supporting this for procedures had to be added. Therefore, the compiler emits a list of the relative addresses of every procedure within the module in the resulting binary. The linker uses this to construct a Table of Contents in the module's data section.

To use this to achieve position-independent code, the overlay manager, upon loading an overlay into memory, fills out the indices of the procedures in that overlay in the table with their current location in memory. When they are evicted from memory, the overlay manager removes them from the table and replace their address with that of the overlay manager itself, indicating that calls must be routed through the overlay manager to load them. Thus, when a procedure currently in memory is called, it will jump directly to its current location in memory; when a procedure currently *not* in memory is called, it will jump to the overlay manager, which will load it.

3.5.2 Position-independence using the overlay manager

Some strategies, such as the one that will be explained in Section 5.5, require additional overhead when using the Table of Contents, to track metadata required

[6]Due to the way this has been implemented in the compiler, it counts instances of the **PROCEDURE** keyword, rather than actual procedures; while this is leads to a slight inefficiency, usage of **PROCEDURE** to indicate a type and not an actual procedure is quite rare in the Oberon system. It occurs only five times in the basic Oberon system, meaning only an additional 20B is accrued due to this simplification.

to support the strategy without being intercepted by the overlay manager. This increases the complexity of the compiler, and has an effect on code size. Therefore, to contrast this method of position-independence with simply using the overlay manager to resolve calls to overlaid code segments, an alternative mechanism was also used. It is effectively a hybrid solution, which maintains a Table of Contents in memory as explained, but does not update it as procedures are loaded into memory and evicted out of memory; hence an overlaid procedure's entry in the table will always point to the overlay manager. The Table of Contents still holds the addresses of procedures that are always resident in memory. In this way, calls to overlaid procedures always go through the overlay manager, while calls to procedures that are always resident in memory do not.

3.6 Placing overlays in memory

Two distinct methods of determining where in memory to place overlays were developed in this thesis. The first, termed a *code buffer strategy*, is to have a dedicated region in memory to which overlays are loaded. This is presented in Section 3.6.1. The second method was to place code overlays on the system heap alongside data; this is detailed in Section 3.6.2.

3.6.1 A code buffer

A very simple strategy for placing code overlays in memory is to create a dedicated memory region for them. In this thesis, this was only done in the case when only one overlay could be loaded at a time — the strategy for keeping multiple overlays in memory simultaneously is detailed in section 3.6.2. As mentioned in section 2.3.2, this will be called a code buffer strategy. This allows for a rather simple overlay manager, as it will always load the requested overlay into the same region every time.

Note that, while it has to keep track of which overlay was overwritten upon a jump, it does *not* have to keep track of the offset into the overlay before it was offloaded. As the same region is always used, the return value will always be correct. For instance, say the overlay region begins at `0x80000`, and a jump located at `0x80604` requires loading a new overlay: prior to the jump, return address `0x80608` is placed on the stack. When the new overlay returns, the old overlay is once again loaded into memory starting at `0x80000`, and as such the stored return address is correct.

3.6.2 An overlay heap

One way of framing the problem of placing code overlays in memory is similar to dynamic memory allocation, i.e. placing code segments where there is space available and deallocating those that can safely be freed. Therefore, implementing a heap for the use of code segments, and not just data, makes sense. Use of a heap

for overlays is similarly done in the proposal for a RISC-V standard for overlays [21].

However, implementing a separate heap for overlays does not come without a fairly steep cost in complexity, as they both have to be maintained. At the same time, implementing two separate heaps in the system brings with it issues of memory fragmentation as well. Therefore, it was deemed more beneficial to use the system heap for code segments. This has some precedent: the early Macintosh OS systems used a similar system for allocating space for system code, such as device drivers, on the heap as they were needed [27]. For those reasons, a similar strategy was implemented here.

Heap allocation on an already full heap: a bug in Oberon

Before integrating overlays into the system heap could be done, an unfortunate bug in the Oberon system that hindered this strategy from working on systems with constrained memory had to be fixed.

In the original Oberon system, a bug in the heap allocator causes the system to crash completely if the heap is full. While this case is rare (but still present) in the original Oberon system, as it has a heap large enough that most programs do not come close to filling it, it is much more noticeable when working with a reduced amount of memory.

The bug is a simple oversight. As explained in section 2.2.4, the heap allocator works with blocks of different sizes in powers of two, and a block of a smaller size is created by splitting a block one size category larger. The procedures for getting a block of size 256B (the largest granularity), and for creating a pointer from an allocated block, both handle the case where a block could not be retrieved due to a full/fragmented heap correctly. That is, if a block could not be allocated, the object is not initialised and a `NIL` pointer is instead returned.

However, the procedures for smaller granularity blocks do not handle this case correctly. An example of one of these procedures, for allocating a 32B block, is given in listing 3.7.

Code listing 3.7: Procedure to allocate a 32B block. If there is an entry in the 32B free-list, that is used; if not, a 64B block is allocated, and the first half is returned while the second half is added to the 32B free-list.

```
1  PROCEDURE GetBlock32(VAR p: LONGINT);
2    VAR q: LONGINT;
3  BEGIN
4    IF list3 # 0 THEN
5      (* block exists in free-list: allocate that *)
6      p := list3; SYSTEM.GET(list3+8, list3)
7    ELSE
8      (* block does not exist in free-list: split a 64B block *)
9      GetBlock64(q);
10     SYSTEM.PUT(q+32, 32); SYSTEM.PUT(q+36, -1); SYSTEM.PUT(q+40, list3);
11     list3 := q + 32; p := q
12   END
13 END GetBlock32;
```

The bug occurs in the case where the 32B free-list is full and no 64B block can be allocated. In this case, a 64B block is unsuccessfully allocated, instead returning 0. As can be seen in line 10, the allocator, in this case, will write to the prefix of the second half of the unsuccessfully allocated block, i.e. $0+32, 0+36, 0+40$. This writes directly into the module table, putting the system into an unrecoverable state that only a complete reboot can fix. (To make matters worse, the first entry of the module table, i.e. address 32, holds a jump instruction to the trap handler; thus, any trap or further attempted allocation would jump to `0x20` to reach the trap handler, only to instead execute the instruction 32, which is not a valid RISC-V instruction.)

The bug is fixed by checking whether the allocation of a 64B block was successful, and only adding the second half of the block to the free-list in the case that it is.

Placing code segments on the system heap

Reusing most of the already extant heap allocator in the Oberon system, the procedure for allocating blocks for an overlay is as follows:

```
1  (* allocates a block of ‘size‘ to heap and writes this address to ‘ptr‘ *)
2  PROCEDURE NewOverlay*(VAR ptr: LONGINT; size: LONGINT);
3    VAR s: INTEGER;
4  BEGIN
5    (* allocate a block on the heap *)
6    IF size = 32 THEN GetBlock32(ptr)
7    ELSIF size = 64 THEN GetBlock64(ptr)
8    ELSIF size = 128 THEN GetBlock128(ptr)
9    ELSE GetBlock(ptr, (size+255) DIV 256 * 256)
10   END ;
11   (* if ptr = 0, then allocation failed. *)
12   IF ptr # 0 THEN
13     SYSTEM.PUT(ptr, size); (* set the tag *)
14     SYSTEM.PUT(ptr+4, -1); (* set mark to -1 *)
15     ptr := ptr + 8; (* return pointer after prefix *)
16     INC(allocated, size);
17   END;
18 END NewOverlay;
```

This procedure allocates the necessary blocks, sets the size and mark, and sets the pointer to after the prefix. Furthermore, the mark is set to -1, which indicates that the block is free; however, because the block is not in the list of free blocks, the heap allocator will never reallocate this block while it is in use. The purpose of this is to communicate to the garbage collector that the block should not be collected, as free blocks are skipped during garbage collection.

Using this procedure, the overlay manager requests a block of the size needed from the heap, then allocates the overlay on top of the allocated space. Most of the Oberon heap allocator can be reused for the purpose of allocating code segments, causing very little code duplication. The process of deallocating code segments from the heap will be explained in section 3.7.

3.7 Garbage collection

This section will cover changes made to the garbage collector in this thesis; it is split in two. Section 3.7.1 motivates the changes made, and covers how the garbage collector was extended to also mark references on the stack, such that garbage collection can run at any time. Section 3.7.2 details the overarching strategy that governs how the garbage collector is integrated into the overlay system.

3.7.1 Marking heap references on the stack

To support running garbage collection at arbitrary times to collect overlays that are no longer in use, a major limitation of its garbage collector must be addressed, namely that it cannot happen in the middle of program execution.

Oberon uses a mark-scan (also known as mark-sweep) scheme [2]. The strategy, in short, is to perform a forest traversal based on a list of roots to mark every discovered pointer as live in the mark phase. Then the scan phase iterates through the entire heap, freeing every unmarked block. Notably, if a pointer is not discovered in the mark phase, regardless of whether it is truly live or not the garbage collector will free it.

Such a thing can occur if the mark phase does not have a complete list of roots. If it does occur, the program whose pointers were not marked will continue to treat its data as allocated, even though it is freed. In such a scenario, if a program requests another block on the heap, this block can potentially be allocated on top of the previous block (which the allocator sees as freed, and the program sees as allocated). This leads to an inconsistency in the heap, as two separate data structures are mapped on top of the same space, with unpredictable results. If one is unlucky enough, a block can be allocated on top of the in-line metadata of another block; if the in-line metadata is overwritten, the heap is permanently corrupted, leading to an irrecoverable state.

Oberon's mark-scan scheme does not take into account pointers that currently reside on the stack, that are either yet to be assigned to a module's static data section or are temporary data allocated on the heap. However, in the original design of the Oberon system, this scheme still provides a complete list of roots, provided the garbage collector is *only* allowed to run when no other programs are running. If a program is in need of more data on the heap, but the heap is full of garbage that can be collected, the program will simply crash rather than invoke the garbage collector. Although this may be an issue within the Oberon system, it does not make itself too well known given enough memory; however, if one wishes to run the Oberon system in an environment with more constrained memory, it is quite detrimental. Ideally, if a program is in need of more memory, the garbage collector should make an attempt to free up enough memory. This becomes doubly necessary if overlays are to be integrated into the garbage collector: if the garbage collector cannot run when needed, stale overlays will never

be collected. Therefore, the garbage collector must be able to be run at any time.

There are several sources of potential roots to blocks allocated on the heap, wherein those relevant to the Oberon system are static data, the stack, and registers [28]. While static data is already dealt with, the stack and registers are not. However, due to a lack of optimisation in the Oberon compiler, any reference to an allocated block on the heap is immediately pushed to the stack upon being allocated and between any use; therefore, pointers in registers can safely be ignored. The stack, however, is a source of potential roots.

There are many potential strategies to efficiently mark the stack. The simplest solution might be to make use of a reference-counting scheme rather than a mark-sweep scheme, as compiler support for this would be quite easy to implement [2], and it would solve the problem of marking the stack entirely. This is not without its downsides, however. Every single operation involving pointers would have increased overhead. Most importantly, however, as noted in [2] and [28], a reference-counting scheme would result in circular data structures never being freed despite being unreachable. This is untenable for the purpose of minimising memory use, as circular data structures would gradually fill up the heap if they were used anywhere in the system. One particular example of where this would frequently occur is in the *Texts* module: text is stored internally as a circular linked list containing fragments of text (i.e. a piece list). Hence, every procedure that deals with text would gradually fill up the heap with garbage that cannot be collected. This could be resolved by explicitly "deallocating" the piece list by breaking the circle at any point, but this adds back the additional responsibility for memory management to the programmer that the garbage collector was initially meant to solve. An alternative approach mentioned in [28] is to use reference-counting next to a separate routine specifically designed to collect cyclical structures. However, this would add additional complexity, and the routine collecting cyclical structures would have to iterate over the stack as well. Therefore, it does not actually solve the problem at hand. For these reasons, using a reference-counting scheme was discarded.

In [28], Shahriyar R. et al. describe a classification of garbage collectors as either *conservative* or *exact*. An exact garbage collector has enough information to construct a list of roots that is certainly complete, while a conservative garbage collector must deal with *ambiguous* references — values that may be pointers to a block on the heap, but may also simply be large integers. If the references are ambiguous, that complicates the procedure of marking them considerably: while the collector must still treat them as valid roots, traversing its tree and marking all other reachable objects, they may also *not* be valid pointers, meaning the collector cannot in the mark phase make any modification to the block it points to, as in the case of an invalid pointer that would corrupt the heap. In other words, its prefix cannot be changed to mark it as live. For the Oberon system, as mentioned, registers can be safely ignored, and an exact list of roots for pointers in the static data section is given. However, references on the stack are ambiguous, and must be dealt with as such. The final implementation can be seen in Code listing 3.8.

Code listing 3.8: Code marking potential references to objects on the heap currently living on the stack.

```
1  PROCEDURE MarkStack*;
2    VAR i, lim, tag, word, p, mark: LONGINT;
3  BEGIN
4    (* look through the whole stack, except for this procedure *)
5    lim := SYSTEM.REG(SP)+28;
6    FOR i := stackOrg TO lim BY -4 DO
7      SYSTEM.GET(i, word);
8      (* check if pointer is on the heap *)
9      IF (word >= heapOrg) & (word < heapLim) THEN
10       SYSTEM.GET(word-4, tag);
11       (* check if the block is marked *)
12       IF tag = 0 THEN
13         (* check tag to ascertain whether it is actually *)
14         (* allocated on the heap *)
15         SYSTEM.GET(word-8, tag);
16         IF (tag >= 100H) & (tag < AllocPtr) THEN
17           p := heapOrg;
18           WHILE (p+8 < word) & (tag > 0) DO
19             SYSTEM.GET(p, tag); SYSTEM.GET(p+4, mark);
20             IF mark >= 0 THEN SYSTEM.GET(tag, tag); END;
21             INC(p, tag);
22           END ;
23           IF p+8 = word THEN
24             Mark(word);
25           END;
26         END;
27       END;
28     END;
29   END;
30 END MarkStack;
```

The solution iterates through every word currently on the stack, searching for values that *look* like pointers to the heap. Then, to disambiguate the word as a potential valid pointer, the heap is scanned, to see if it truly does point to a block on the heap. If it turns out to be a valid pointer to the heap, then the block it refers to is marked as live. As scanning the heap is quite expensive, deciding whether a value looks like a pointer consists of several steps. It first checks if it points to the heap at all. Then it checks if the block on the heap is unmarked — if it is marked, it has already been handled previously in garbage collection, and can safely be ignored. Finally, it checks if the object's type tag points to anywhere in the static data section, as if it doesn't, it cannot be a valid block. Note that this still is not an exact strategy: a value on the stack may by coincidence point to a valid block on the heap, despite not being a pointer. However, this case is rare enough to be tolerable, and the only side effect caused by this limitation is that some garbage on the heap takes longer to be collected.

With this implementation, there is only one case not accounted for, which is valid pointers that point into the middle of an allocated block. As noted in [29], however, such cases are usually quite rare. Furthermore, the original implementation of garbage collection in Oberon makes no attempt to mark such objects, and no code that would call for supporting this edge case appears in the ori-

ginal Oberon operating system. Additionally, the Oberon programming language dissuades such arbitrary pointers, by forcing all pointers to be typed outside of low-level code (that is, code which makes use of the low-level SYSTEM module). As such, it is considered non-standard Oberon code, and can safely be ignored.

This solution is integrated into garbage collection by being called after all pointers in static data have been marked. This allows it to finish more quickly, as it can skip past valid pointers on the stack that have already been marked.

3.7.2 Using garbage collection to free overlays

In section 3.6.2, which detailed the procedure for allocating code segments on the heap, overlays had to incorporate a two-word prefix to avoid breaking garbage collection; to that end, the tag of all allocated overlays was set to -1. However, this mark can also be used to communicate with the garbage collector whether the block in question should be collected or not.

A downside to the way the Oberon garbage collector works — which limits the elegance of this scheme somewhat — is that free blocks have their size indicated by a literal value, while allocated blocks contain a pointer to the type descriptor. Thus, while ideally a strategy would only have to set the mark to 0 when that overlay is deemed to be garbage that can be collected, it would also have to change the tag of the block to be a pointer to the overlay's size on the heap rather than its size as a literal. However, by changing the mark and the tag in this way, it can successfully be communicated to the garbage collector.

With this complete, garbage collection can run at any point in program execution. This is done by checking, whenever data or code is allocated on the heap, the allocation succeeded; if not, garbage collection runs, and allocation is attempted again. If allocation still does not succeed, the heap is too full to currently contain the data/code; for data, this means returning a NIL pointer, and it is the program's responsibility what to do with it, whereas for a code overlay, execution cannot proceed if it cannot be loaded into memory, and as such the Oberon system instead stops the program and resets (in exactly the same way as when a program encounters a trap).

3.8 Module-granularity overlays

The first version developed focused on achieving overlays that acted on the granularity of modules. In this strategy, each overlay consists of an entire module. When the overlay manager is given a call to a module that currently is not loaded into memory, that overlay will be loaded, and the Table of Contents will be filled out with entries to that module's procedures. When the loaded module is evicted, the Table of Contents is filled out with entries that all point to the overlay manager, such that a call to a procedure within the module will be routed through it. This allows for a rather small overlay table, as the system only has to consist of nine overlays in total to boot; however, some additional overhead to keep track of the

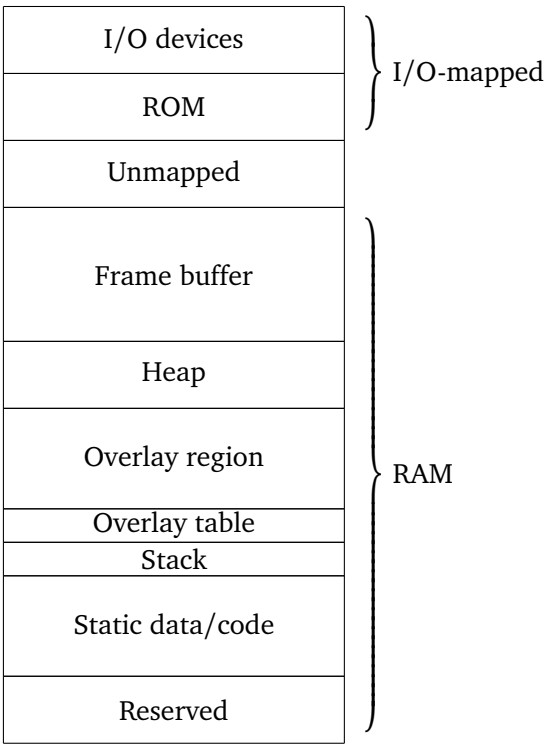


Figure 3.3: Memory layout of the Oberon system when using a code buffer strategy. The size of the boxes represents approximate relative use in memory.

procedures within a module, to update the Table of Contents and correctly branch, is required. This granularity was implemented using a code buffer strategy only; the resulting memory layout can be seen in Figure 3.3.

An overlay table is constructed using the record seen in listing 3.9, occupying 11 bytes in total; to preserve word-alignment, the last byte is padded, making it 12B.

Code listing 3.9: The record containing information of overlays given a module-granularity strategy.

```
1 OverlayDesc = RECORD
2   procs: Procedure;
3   sector: INTEGER;
4   size: BYTE;
5   mapped: BOOLEAN;
6   mno: BYTE;
7 END;
```

Entry `procs` points to a list that describes procedures in the module, allocated on the heap. `Sector` indicates which sector on the disk the overlay's code starts, `size` indicates how large the overlay is at 256B granularity. `mapped` indicates whether the overlay is currently in memory or not. `Mno` indicates the overlaid module's location in the module table. This is used by the linker when linking

modules that depend on the overlaid module, such that it can identify which overlay index to fixup overlay tokens to. The specifics regarding changes made to the linker are given in section 3.3.

Note that, while the `mapped` field is largely unnecessary as the currently mapped overlay can also be indicated using a global variable, it is slightly more performant to maintain whether a specific overlay is mapped or not than to track it as global state. Additionally, it makes no difference in memory footprint of the overlay table: if it were removed, an entry would still occupy 12B due to word-alignment.

There are a few additional tasks the overlay manager must perform to correctly work in this strategy. Firstly, the overlay manager must track which overlay to load when returning from an overlay code segment, as discussed in Section 3.9.1. Secondly, it must branch to the correct procedure within a module. This was done by including the index of the procedure being branched to in the overlay token, since this is also used for fixups (see Section 3.3). This does incur some overhead, as the overlay manager must find the address of the correct procedure by investigating stored metadata, and then branch to it. While this could be made more performant, this strategy did not have promising results for the Oberon system (see Section 5.4 and Section 5.6); as such, rather than refine it further, it served as the baseline for later strategies, which will be explained in Section 3.9.

3.9 Function-granularity overlays

Two different strategies were implemented for function-granularity overlays: one using a code buffer approach in Section 3.9.1, and one using the system heap in Section 3.9.2. Additionally, the strategy using the system heap was implemented with both the approaches to position-independence presented in Section 3.5. Both approaches create one overlay per procedure.

3.9.1 A code buffer

Using a code buffer for function-granularity overlays is very similar to the approach taken with module-granularity overlays. However, the major differences will be highlighted here.

Firstly, the Table of Contents does not need to be updated, as only one function is kept in the code buffer at any time; the overlay manager handles recursion to avoid loading the same code segment on top of itself. Secondly, the overlay table has to be considerably larger in terms of number of entries, though each entry is smaller, and does not also have to allocate additional information on the heap. The specific information stored in the overlay table can be seen in Code listing 3.10.

Code listing 3.10: The record containing information of overlays given a function-granularity code buffer strategy.

```
1 OverlayDesc = RECORD
2   sector: INTEGER;
3   size: BYTE;
```

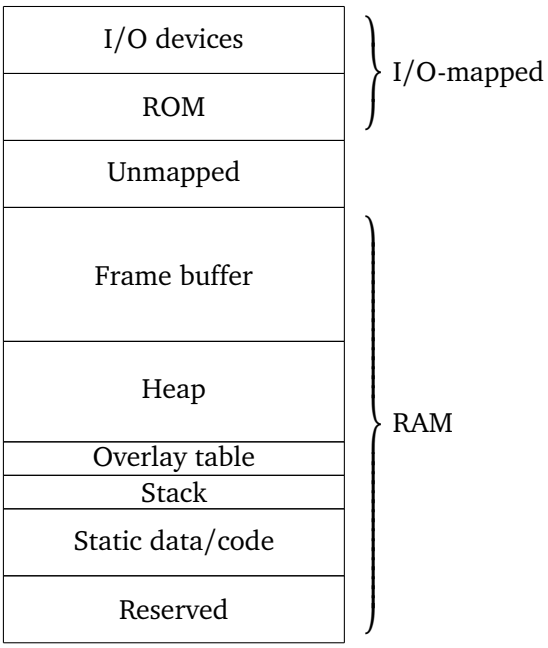


Figure 3.4: Memory layout of the Oberon system, modified to support function-granularity overlays on the heap.

```
4   mapped: BOOLEAN;
5   mno: BYTE;
6   pno: BYTE;
7 END;
```

This record fits into 8B, using the same strategy of storing the overlay's size at 256B granularity. Both `mno` and `pno`, the index of the module into the module table and the procedure into the module's procedure table, are both stored for the linker to be able to identify the correct overlay index when fixing up overlay tokens.

3.9.2 An automatic overlay tree strategy on the heap

Placing code segments on the heap allows multiple different code segments to be live simultaneously, rather than only one, as in a code buffer strategy. In addition, garbage collection has been integrated into this scheme such that code segments that are not considered live can be collected in exactly the same manner that data that is no longer considered live is collected. The memory layout of the Oberon system when this strategy is used can be seen in Figure 3.4.

However, the question of how to ascertain which code segments are to be considered live still needs to be addressed. To address that, the strategy chosen was to generate an overlay tree (see Section 2.3) on the heap, using the marks in a heap block's prefix to indicate whether a block is currently in a call tree or not.

Thus, the goal of this strategy is to create an overlay tree for any potential call path that can be encountered, without requiring static analysis of the call graph beforehand.

The basic strategy is as follows. Recall that the garbage collector never collects blocks with a negative mark, regarding them as already free, and the heap allocator will never allocate a block with a negative mark if it isn't in the free-list. When an overlay is first loaded, its mark is set to -1, as before; then, whenever this code segment is called, its mark is subtracted by one. Upon returning from the allocated code segment, its mark is once more increased by one. Hence, whenever it is actively in the code tree, it will have a mark below -1, and will not be collected. Whenever garbage collection runs, all overlays whose mark is -1 will have their tags replaced with a pointer and their mark set to 0, marking it as garbage. (If it is needed again later, it will have to be loaded anew.)

There are several positives to this strategy. Firstly, its overhead is rather low: incrementing/decrementing the mark is a simple routine. Furthermore, upon running the garbage collector, the only extra consideration that needs to be made is for the overlay system to be updated with regard to which overlays were evicted.

Notably, it would be more elegant if the mark started at 0, such that it would be considered garbage without any further intervention. However, due to the way the garbage collector works, this would not work. While it would be possible to further modify the garbage collector to support this, it is considered outside the scope of this thesis.

Additionally, compiler support was needed to update the mark when using the strategy for position-independence presented in Section 3.5.1. In the case that it fetches the current address of a loaded procedure from the Table of Contents, that procedure's mark will have to be decremented. To that end, the compiler has to insert instructions that load the mark, decrement/increment it, and then store the changed mark, before jumping to the procedure and after returning. With the strategy presented in Section 3.5.2, no such compiler support is needed: as every call to an overlaid procedure goes through the overlay manager, it can be handled in the overlay manager rather than by the compiler. (Note that, due to inefficiencies in the compiler described in Section 2.2.3, this will unavoidably be less performant than direct compiler support, though with the benefit of smaller code in overlaid procedures.)

Chapter 4

Methodology

The methodology of this master's thesis is similar to the one developed in our previous report [1], with regard to what data is available and how it was gathered.

This chapter is laid out as follows. In Section 4.1, the environment in which the tests are run — a RISC-V emulator — is described. This section also includes info on all the data that can be gathered within this test environment. Section 4.2 describes the limitations of the chosen methodology, i.e. the data that *cannot* be gathered in this testing environment. Finally, Section 4.3 describes the specific methodology for executing test programs within this testing environment.

4.1 Testing environment

This section will first detail the testing environment itself, and then the specific data that can be gathered within it.

4.1.1 RISC-V emulator

Evaluation of the operating system has been performed on a RISC-V emulator. This emulator is a fork of another emulator, written by Peter de Wachter[1], which was extended to support RISC-V[2] in my previous project, detailed in [1]. The RISC-V emulator was extended by integrating an already extant emulator into it, namely Ted Fried's compact RV32I emulator[3].

4.1.2 Files included in the testing environment

It is worth noting that the performance of many programs is considerably impacted by the number of files registered in the file system. As the testing envir-

[1]Peter de Wachter's emulator can be found here: `https://github.com/pdewacht/oberon-risc-emu/`

[2]The Oberon system emulator supporting RISC-V can be found here, along with a disk image of the RISC-V Oberon system: `https://github.com/solbjorg/oberon-riscv-emu`

[3]Ted Fried's emulator can be found here: `https://github.com/MicroCoreLabs/Projects/blob/master/RISCV_C_Version/C_Version/riscv.c`

onment allows for quickly building different images including different files, only the files required to perform a specific test, along with their source and symbol files, are included in any image, to minimise any noise between tests that this might introduce.

4.1.3 Gathering data on instructions run

To gather data on number of instructions run during certain workloads, the Oberon compiler was augmented with features for debugging and measuring performance. An `EBREAK` procedure was added to the low-level `SYSTEM` module, which simply emits an `ebreak` instruction. Whenever the emulator executes an `ebreak` instruction, it enters an interactive debug mode, where, among other implemented features, a count of instructions run can be started. (The count ignores any `ebreak` instructions.) In this way, by placing an `ebreak` instruction at the beginning and end of the workload being measured, the number of instructions the emulator executes is gathered.

4.1.4 Memory usage

Data regarding the stack is collected by monitoring the stack pointer: by comparing its current position to base of the stack, the current size of the stack can be ascertained, and the necessary maximum size of the stack required for any given testbench can be found.

Data regarding the heap is collected in the emulator, by scanning the heap in the same manner as the Oberon system, and outputting details regarding allocated/freed blocks. Free-lists are similarly investigated.

4.2 Limitations of the testing environment

As the tests are run in an emulator, there are necessarily limitations with regard to what data can be gathered from it. As such, while it is easy to gather data on binary size and instructions run, some of the data that cannot reliably be gathered by the emulator will be described in this section.

4.2.1 Disk write/read performance

While the emulator does emulate the behaviour of reading data over SPI (as is done in the original Project Oberon [2]), it does not cover the impact reading from the disk might have on performance beyond the instructions required to send/receive SPI signals, instead assuming that the read/write to/from memory happens instantaneously. Although the characteristics of the disk could be modelled in the emulator more accurately, it would be hard to glean meaningful data from it. The

original Oberon system uses an SD card for this purpose, and as such the performance characteristics would be considerably impacted by the specifications of the chosen SD card [30].

4.2.2 Memory performance

Similarly, memory performance is not accurately modelled, instead working off the assumption that they happen within a cycle. Although on many hardware platforms this would be an unreasonable assumption, in this case it is not that notable: the Project Oberon 2013 system ran entirely on the Spartan-3 Starter Kit Board with the use of SRAM [2]; as such, memory operations only take two cycles [31]. While this could be modelled to an extent in the emulator, this data would not be particularly useful for several reasons, detailed in Section 4.2.3.

4.2.3 Cycles spent

This methodology chapter has been careful to consider only *instructions run* rather than *clock cycles*, as the latter are hard to measure without making more assumptions about the hardware platform. For instance, the original RISC-5 processor used two cycles for multiplication, although it only occupies one instruction [11].

For the purpose of this thesis, an exact number of cycles run is not available to us. While this can be estimated by modelling the cycle latencies of each instruction in a specific hardware platform, this was not done for this thesis, as it would be hard to glean meaningful data from it. Putting together the potentially disparate latencies of e.g. multiplication and division, in addition to the potential latencies in memory operations, the amount of assumptions about the platform that must be made are many enough as to render the results meaningless for the purposes of evaluating its performance on any particular actual platform.

At the time of writing, the Oberon system ported to RISC-V has not yet been tested on hardware. As such, making assumptions about the performance of the underlying hardware platform is difficult.

4.3 Test methodology and experiments

This section will detail the specific methodology regarding how particular aspects of the Oberon system are tested. Section 4.3.1 will cover gathering data with regard to the code size of Oberon modules, Section 4.3.2 will cover measuring the efficiency of loading data from secondary storage to memory, and Section 4.3.3 will detail the tests run on the changes made to heap allocation and garbage collection.

4.3.1 Code size

As additional glue code has been added in different ways in different strategies, it is interesting to note the effect these changes have on total code size. This has been gathered by compiling a set of files under different setups, and outputting the total size of the *compiled code section*. Note that this does not include increases in header size of the binary, nor additional data stored in-memory, but only code. This choice is made because more information is included in the header of binaries compiled with overlay support, without this directly affecting the size of the static data section or the outputted code, and the amount of space a file occupies on secondary storage is not of interest in this thesis.

To get a broad range of data, a variety of applications were chosen. Firstly, the Oberon compiler, which to our knowledge is one of the largest programs written for the operating system (about 8000 words smaller than the operating system itself in terms of code on the RISC-5 [2]); the RISC-V compiler ported in our previous project [1] was used, simply because it is more relevant to this project than the RISC-5 compiler. Secondly, the full Oberon operating system, i.e. everything required to boot it. This includes the inner core, and the entire outer core, as explained in Section 2.2.4; both the size of the entire system as well as the inner and outer cores on their own are gathered. These files are very relevant to the project: the size of the inner core determines how much code must always be resident in memory, and the size of the outer core determines how much code will be overlaid. Finally, Hilbert, Sierpinski, and Checkers are small toy modules that nonetheless demonstrate basic parts of Oberon programs such as frame handlers.

4.3.2 Data transfer between disk to memory

As explained in Section 3.4, transfer of data from secondary storage to main memory is expected to be the bottleneck of the system, and was therefore made more efficient; as this is a particularly important component in making code overlays work on the Oberon system, the results of this optimisation are presented separately.

To test the efficacy of the method for transferring data from disk to memory presented in Section 3.4, a test harness on top of an otherwise unmodified Oberon system was built, to test it without interference from other parts of the system (e.g. overlays or position-independent code). The test harness allocates a block on the heap to write to, and then has two tests: one for reading starting from a specified sector with a specified size to the block, and another for reading from a file to the block. The test harness can be found in Code listing A.3.

A selection of files was chosen to test the efficacy of the two approaches both for smaller segments of data as well as somewhat larger segments of data, which can be seen in Table 4.1. Two files, File1.txt and File2.txt, with exactly the same size but differing contents were included, to determine if the contents of files have any impact on read performance. File3.txt was included to test difference in performance when a file only barely crosses a block boundary (as the block size

Table 4.1: The sizes of files on which read performance was tested.

File	Size (in bytes)
Blink.Mod	320
Sierpinski.Mod	3157
Hilbert.Mod	2497
File5.txt	256
File4.txt	1025
File3.txt	513
File2.txt	512
File1.txt	512

is 512B). Similarly, File4.txt was included to test difference in performance when a file barely crosses a sector boundary, as the sector size is 1024B. File5.txt was included to test performance when a file is significantly smaller than a block, as the rest of the block will still have to be read, as noted in Section 3.4. Finally, the remaining Mod files, Hilbert, Sierpinski, and Blink, were chosen to give an impression of performance on "regular" data.

4.3.3 Dynamic memory allocation

As explained in Section 3.7, changes were made to garbage collection to support collecting overlays placed on the system heap. However, as mentioned, it also has the positive benefit of allowing garbage collection to run whenever more space is needed on the heap. To capture the precise effect this alone has on the memory footprint, this will also be tested separately. This will only be tested in isolation on the phases of booting the Oberon system.

To test this, the changes required to mark pointers on the stack and run garbage collection whenever needed were added to an otherwise unmodified version of the RISC-V port. Then, for both the version with garbage collection and the one without, the heap was constricted as much as possible such that the system could only barely successfully boot, down to 256B granularity.

4.3.4 Testbenches

To test overlay strategies on specific programs, a generic test harness was written. The test harness works by using System.Tool — a text file meant to act as a menu listing commands — to indicate the specific test that should be run after the operating system is done booting. For instance, if System.Tool contains only the text PrimeNumbers.Generate 25, then that test will be run. The test is run using the procedure *TextFrames.Call*, which is usually invoked by a series of other procedures after the user presses the middle mouse button on a command. It is done this way so that programs that scan parameters do not require modification to run correctly. After being called but before being run, each test resets the

stack pointer, evicts overlays and runs garbage collection, to make the isolated testbench as consistent between setups as possible.

For heap-based strategies, each experiment is run on four different setups: firstly, a normal Oberon system, without overlays; then, a warmed up system, where the heap is large enough to fit the working set of the application, and it has been run once beforehand such that all overlays are already loaded into the heap; then, a system that hasn't been warmed up, such that code segments are loaded into the heap as needed; and finally, a system with a more strained heap, defined as the smallest heap possible to boot the system and run the test. Furthermore, to capture how the performance of the program scales with a larger workload, each setup was run several times for a differing number of primes, as can be seen in the figure. For these experiments, overlays were evicted and garbage collection was run before running the experiment, as otherwise, a few overlays would somewhat unpredictably be resident in memory prior to running[4].

For strategies using a code buffer, the experiment is run on two different setups: a normal Oberon setup, and one using overlays.

Finally, while setups have been tested with restricted amounts of memory, note that the experiments are all performed on the same emulated system as the original RISC-V port, which is to say with 1MB of memory. This is for the sake of keeping the testing environment consistent. However, their memory layouts, particularly with regard to the heap, are still modified (by changing parameters set in the Oberon kernel) to perform different experiments, such as by shrinking the heap.

PrimeNumbers

One of the experiments run is to generate a variable number of prime numbers using the module PrimeNumbers, which can be seen in Code listing A.2. PrimeNumbers is a rather simple module that generates a user-specified number of primes, and outputs them to the system log after it is complete.

There are two important things to note about this test. Firstly, while the module itself only contains two procedures, of which only one, *Primes*, does most of the work in the module, that procedure includes calls to a module present in the outer core, **Texts**. Thus, if **PrimeNumbers** is used as a test, it does not test overlaying only two procedures as it might seem: it tests a total of 31 procedures. As the entire outer core of the operating system is overlaid, the procedure calls in *Generate* to module **Texts** involve *many* other procedures. Most notably, the call to `Texts.Append`, which appends the text buffer to the System Log text frame, involves several layers of procedure calls to correctly update and render both the

[4]Note that for an overlay tree strategy, evicting overlays before running the main program allows for a more minimal heap, as the OS procedures necessary to call the main program no longer have to be resident in memory. To do this and avoid crashing due to overlays higher up in the tree no longer being resident in memory upon returning, the program re-enters the Oberon main loop upon finishing execution. This can be done by any program with a clearly defined end point upon which it is supposed to return control to Oberon.

Table 4.2: A table containing details regarding the PrimeNumbers.Generate program.

Call depth	11
Stack size	544B

menu and the text; this call also has the largest call depth, of 11 procedures. Secondly, as a larger number of primes is generated, not only does the loop generating prime numbers start to occupy a larger proportion of the runtime, but so does the phase that appends the text. Important details regarding PrimeNumbers are given in Table 4.2.

Hilbert

The other experiment is to draw a Hilbert curve using the Hilbert program, which can be seen in Code listing A.1. In doing so it opens a viewer on the left side of the screen; if another program with a viewer is opened, the Hilbert curve will be resized to take up half the screen, as the other program takes up the other half. For this test, it will only be tested with drawing on the left side of the screen.

Table 4.3: A table containing details regarding the Hilbert.Draw program.

Call depth	14
Stack size	340B

This test is interesting as it is uses mutual recursion, which requires many indirect function calls, as well as many small procedures. Furthermore, it tests aspects of the underlying Oberon system, such as creating Viewers and Handlers. As such, while this program is simple, it is able to test the effect of overlays on programs that run on top of the Oberon system. Important details regarding Hilbert are given in Table 4.3.

4.3.5 Booting

A large experiment is to boot the entire system. This experiment is considerable, as it involves initialising the inner core; initialising, linking and generating overlays for the first phase of the outer core; and initialising, linking and generating overlays for the second phase of the outer core. This encompasses creating overlays from a total of 197 procedures, contained in 9 modules, as well as initialising them, including system viewers.

The experiment is considered from when execution first starts in RAM — i.e. the bootloader, located in the ROM, has finished executing, and transferred execution to the inner core — until the main Oberon loop is entered. The experiment is successful if it enters the loop with everything initialised correctly. Finally, note that it boots with the System.Tool viewer — which contains a set of useful commands presented to the user on startup — but the file System.Tool is empty, to

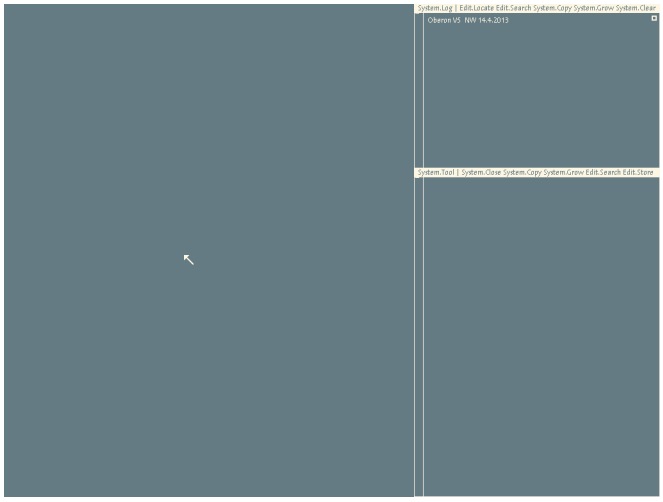


Figure 4.1: The Oberon system, when fully booted according to the methodology presented in Section 4.3.5.

avoid the size of System.Tool impacting the time spent booting. Finally, to avoid the number of unnecessary files included impacting the time it takes to boot, as noted in Section 4.1.2, results are gathered with only the files necessary to boot included, along with their source and symbol files. For the sake of clarity, the state of the Oberon system when it is considered fully booted is presented in Figure 4.1.

In addition to the number of instructions required to fully boot the system across different setups, information on how much memory is required to boot the system is also gathered, for the purpose of contrasting the performance of different strategies against their memory footprint.

Chapter 5

Results and Discussion

In this chapter, results related to the different components built to support code overlays will be presented, explained, and discussed. In Section 5.1, results related to the system created to move data from the disk to memory will be presented and discussed; the same will be done for the strategy for position-independence in Section 5.2. Section 5.3 presents the incidental effect of the changes made to the garbage collector, namely a decrease in the minimal heap size required to run the Oberon system. Sections 5.4 and 5.5 will present results related to the different overlay strategies developed in this thesis. Finally, Section 5.6 presents the overarching effect these different strategies had on actually being able to run the system itself.

5.1 Transfer of data from disk to memory

In this section, results regarding the changes made to the files system are given, in particular the difference between loading a file via the file system versus from the disk without any intermediary steps.

The results for reading are given in Figure 5.1. On average, it takes 1090% more instructions to read from the file system than directly from disk.

There is a sizeable difference in efficiency between loading overlays into memory through the file system and directly from the disk. Therefore, the need for such a system has been made clear. A few things are of note, however.

Firstly, differences in the contents of files makes no difference in the time it takes to load them, as can be deduced by the results of loading File1.txt and File2.txt. Secondly, an interesting result is visible between File2.txt and File3.txt. Note that File2.txt is 512 bytes, and File3.txt is 513 bytes. File3.txt incurs a significant hit on performance for reading directly from the disk (67.5% increase in instructions run), while the behaviour when reading from the file system is largely unchanged. While this seems like it would be a negative, it is actually a positive: the reason this behaviour is exhibited is that the file system always reads full sectors, of size 1024B, while the alternative approach created from this thesis reads

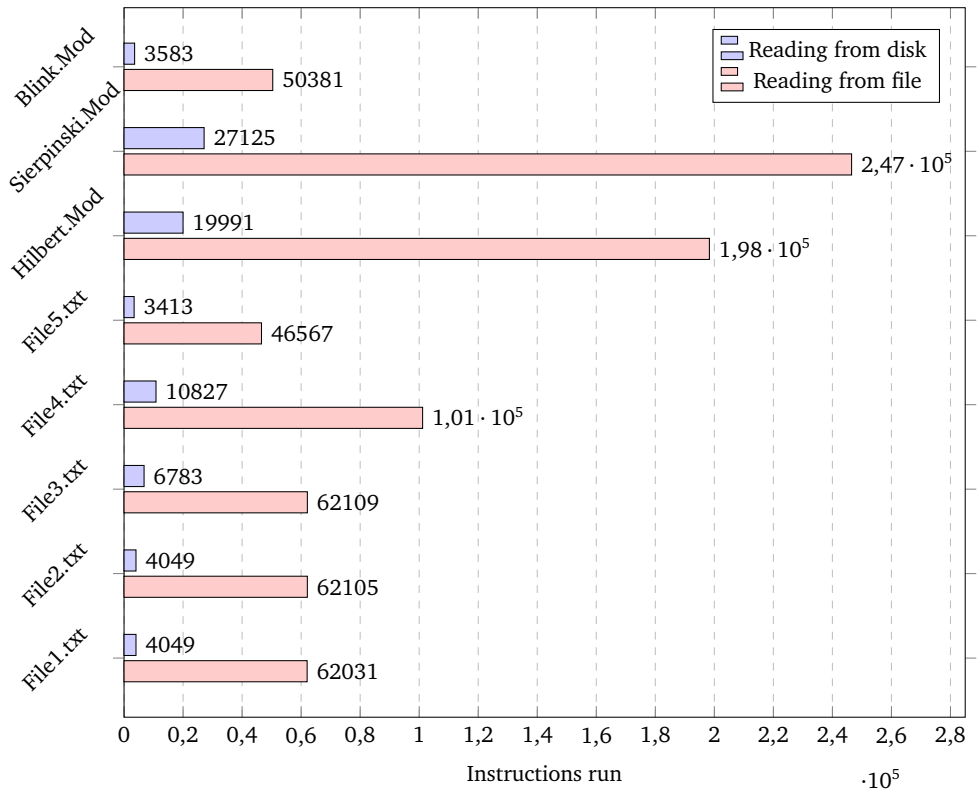


Figure 5.1: Instructions required to load the files given in Table 4.1 from secondary storage into memory, when loading through the file system or directly from disk. Fewer instructions indicate better performance.

Table 5.1: Overhead in code required to support position independence using a Table of Contents.

Module	Size without ToC (B)	Size with ToC (B)	% increase
Hilbert	2 652	2 768	4.37%
Sierpinski	3 176	3 384	6.55%
Checkers	1 324	1 396	5.44%
RVOP	29 300	30 956	5.65%
RVOS	6 792	7 308	7.60%
RVOB	12 160	12 796	5.23%
RVOG	34 392	40 468	17.7%
Inner core	34 108	36 300	6.31%
Outer core	87 416	93 032	5.61%
Minimal Oberon system	122 320	129 340	6.06%

only as many blocks as is necessary from each sector. Thus, the file system sees no difference in performance between reading files of sizes 512B and 513B because it is wasteful in reading the former. This can be seen when comparing the performance in reading File3.txt and File4.txt: both approaches take a proportionally similar hit in performance (62.9% and 59.6% for file system and directly from disk respectively), as they both have to start reading from a new sector.

Thirdly, File5.txt sees less time spent in reading it than File1.txt and File2.txt, which is to be expected, as it is half the size. However, due to overhead in both reading directly from disk and from the file system, it is not half the time, for reasons that have been discussed earlier. It should be noted that despite the file system reading the whole sector here, too, the time spent in reading from it is still significantly less, as the actual time spent reading the file into memory byte by byte has been cut in half.

While this strategy is far more efficient for the use case of supporting code overlays, it is not nearly as useful for more general-purpose applications. This performance is achieved by bypassing all high-level overhead that are unnecessary for linearly reading a segment of data from a known sector with a known size — which is not the use-case of most applications.

5.2 Position-independent code

As mentioned in Section 3.5.1, a total of 1172B are expended in storage overhead to hold the Table of Contents for every module in the Oberon system. In addition, it leads to additional code size for making function calls, as can be seen in Table 5.1. (Note that the version of the compiler given in the table is of the original Project Oberon port to RISC-V, and not the compiler with support for table of contents and overlays.)

Among the files chosen — which should be quite representative, as one sample includes the entire Oberon system — the mean value of the code overhead of the

Table 5.2: Minimum heap size required for different workloads depending on whether stack marking is included.

Testbench	Without stack marking	With stack marking	% reduction
Inner core of the original Oberon system	8 448 B	8 448 B	0%
Outer core (phase 1) of the original Oberon system	26 368 B	17 664 B	33.0%
Full original Oberon system	41 472 B	27 136 B	34.6%

Table of Contents is 6.57%. While most files see a reasonable increase in code size, the increase of the compiler's code generator, RVOG, is considerably more extreme. The reason for this is fairly straightforward: a very large portion of the code generator consists of procedure calls, in particular procedure calls to encode instructions and then place that instruction in the code buffer. Thus, it is by far the most extreme case available in the Oberon system when it comes to this particular change. Despite this, most files see less than half that proportion of increase, with the Oberon system as a whole — which is the most important to this thesis — seeing an increase of 6.06%. Note, however, that this only includes code required to branch using the Table of Contents, and not additional code required to perform the strategy.

Note that it would be very feasible to make the inner core not use the Table of Contents, as it is never overlaid. For the sake of keeping the compiler simple, this was not done in this thesis, however it would not be particularly difficult to implement should one desire to do so.

The performance of the Table of Contents, as well as contrasting it with the alternative proposed in Section 3.5.2, is not presented here, as they are not meaningful without the overlay system for which this was built. Therefore, that will instead be discussed in Section 5.5.

5.3 Dynamic memory management

The minimum heap size required to boot the Oberon system is shown in Table 5.2, contrasting the difference with and without being able to run garbage collection while booting. As explained in Section 3.6.2, garbage collection cannot be run in the middle of program execution without a scheme for marking references to the heap on the stack. As can be seen, over 10kB of heap space is saved in the full Oberon system with the inclusion of stack marking.

As noted in Section 3.7.1, this is a conservative garbage collector with respect to the stack: while the list of pointers in modules' data sections is known exactly, the list of pointers to the heap on the stack is not. As mentioned, this could have some impact on the amount of space possible to save on the heap. However, only

six references are actually marked while booting the full system, all of which are pointers: the stack marking procedure encounters no false positives. Thus, while an exact garbage collector might be more performant, it would not save more memory while booting the system.

Another thing of some note is that this makes absolutely no difference with regard to booting the inner core. The reason for this is simple: the only record allocated on the heap in this initialisation process is the sector map[1] — a structure that occupies 8192*B*, and will never be garbage collected. The heap must be 8448*B*, however, to accommodate for the inline metadata. Even though the inline metadata is only 8B, due to the design of the heap allocator as described in Section 3.6.2, one cannot mix the granularity of the blocks being allocated for a record. As such, an additional 256B is required.

An issue that has not been discussed here is the effect this solution has on heap fragmentation. This will be discussed in more detail in Section 6.2.

5.4 Module-granularity strategies

This section focuses on results gathered using overlays operating entirely on the granularity of modules. Results from the experiments run on this overlay strategy can be seen in Table 5.3. The first column indicates the testbench being run, and the second column indicates how many times an overlay (i.e. a module in this case) is loaded into the code buffer. The third column indicates how much code is, in total, loaded into memory, and the fourth column indicates how much additional space is required on the stack compared to the version of the program running on a standard Oberon system (which can be found for both testbenches in Section 4.3.4). The fifth and sixth columns indicate how many instructions are required for the program to finish on a standard Oberon system and in the overlaid system respectively, and the seventh column indicates the percentage increase in instructions run.

As can be seen, there is a very large performance degradation in the use of module-granularity overlays. Notably, however, there is a very large difference in the performance degradation of Hilbert and PrimeNumbers. This scales in large part with how much code they have to load into memory, which is an order of magnitude more for `PrimeNumbers.Generate 25`. This is in large part due to a single source of interference in the Oberon system, that PrimeNumbers is affected by and Hilbert is not.

The interference in question is found in the procedure `TextFrames.DisplayLine`, which can be seen in Code listing 5.1.

Code listing 5.1: The *DisplayLine* procedure, found in the module **TextFrames**.

```
1 PROCEDURE DisplayLine (F: Frame; L: Line;
```

[1] In the Oberon system for RISC-5 it is not stored on the heap, but in the static data section. It was moved to the heap due to limitations in how much data can be accessed in the static data section, as detailed in [1], which can be found in Appendix B.

Table 5.3: Performance of module-granularity overlays, using a code buffer, compared to a normal Oberon system.

Testbench	Overlay loads	Overlaid code (B)	Additional stack use (B)	Instructions (no overlays)	Instructions (overlaid)	% increase (instructions)
Primes.Generate 25	652	13 481 984	208	71 371	104 328 776	146 078
Primes.Generate 50	1 364	28 313 088	208	156 019	219 082 646	140 320
Primes.Generate 75	2 076	43 144 192	208	243 529	333 839 106	136 984
Primes.Generate 100	2 794	57 748 830	208	332 549	448 597 767	134 797
Primes.Generate 125	3 515	72 628 335	208	438 793	564 186 223	128 477
Primes.Generate 150	4 224	87 401 505	208	514 550	678 972 478	131 855
Hilbert.Draw	11 250	51 532 440	204	1 616 519	430 005 061	26 500,7

```
2   VAR R: Texts.Reader; X, Y: INTEGER; len: LONGINT);
3   VAR patadr, NX,  dx, x, y, w, h: INTEGER;
4 BEGIN NX := F.X + F.W;
5   WHILE (nextCh # CR) & (R.fnt # NIL) DO
6     Fonts.GetPat(R.fnt, nextCh, dx, x, y, w, h, patadr);
7     IF (X + x + w <= NX) & (h # 0) THEN
8       Display.CopyPattern(R.col, patadr, X + x, Y + y, Display.invert)
9     END;
10    X := X + dx; INC(len); Texts.Read(R, nextCh)
11  END;
12  L.len := len + 1; L.wid := X + eolW - (F.X + F.left);
13  L.eot := R.fnt = NIL; Texts.Read(R, nextCh)
14 END DisplayLine;
```

The procedure in question displays a given line of text on a text frame in Oberon, and is for instance used for writing to the log, writing a file's contents to the screen, etc. This procedure displays a high level of interference within the **WHILE** loop from line 5 to 11, in the pattern of *Fonts, TextFrames, Display, TextFrames, Texts, TextFrames*, which occurs for every character to be written to the text frame. For example, writing the word "Oberon" onto a text frame would require iterating through the loop six times, representing a total of 36 loaded overlays. Although one might consider using alternative strategies to better deal with this, the success of such strategies is severely limited at module-level granularity. This is particularly the case for this example, as *TextFrames* is the largest module in the base Oberon system, occupying a total of 34948B. Even with a heap-based strategy, to avoid this interference, all four modules would have to be loaded onto the heap, occupying a total of 57156B on the heap — thus being very inefficient with regard to the memory footprint.

Note that the performance degradation with regard to Hilbert is still considerable. Although a large part of Hilbert consists of procedure calls within the same module, it is still very slow under this strategy for two reasons: firstly, these procedure calls are predominately using indirect procedure calls, which are unable to exploit the Table of Contents; and secondly, every time Display is called to draw

Table 5.4: Performance of function-granularity overlays using a code buffer, compared to a normal Oberon system.

Testbench	Overlay loads	Overlaid code (B)	Additional stack use (B)	Instructions (no overlays)	Instructions (overlaid)	% increase (instructions)
Primes.Generate 25	939	683 520	212	71 371	7 289 216	10 113
Primes.Generate 50	1 909	1 370 624	212	156 019	14 636 942	9 281,5
Primes.Generate 75	2 879	2 057 728	212	243 529	21 987 177	8 928,6
Primes.Generate 100	3 851	2 746 880	212	332 549	29 376 403	8 733,7
Primes.Generate 125	4 831	3 444 736	212	438 793	36 817 701	8 290,7
Primes.Generate 150	5 801	4 131 840	212	514 550	44 187 540	8 487,6
Hilbert.Draw	29 593	15 184 995	200	1 616 519	181 978 298	11 157

a line of the curve, it has to be loaded, draw a line, and then load Hilbert back.

5.5 Function-granularity strategies

In Section 3.9, two overlay strategies using function-level granularity overlays were developed. The first used a code buffer that can hold a single procedure at a time, while the second developed on it by placing these procedures on the system heap, allowing multiple procedures to be in memory simultaneously. The results from the former will be presented and discussed in Section 5.5.1, while the latter will be presented and discussed in Section 5.5.2.

5.5.1 Code buffer results

The performance of the function-granularity overlay strategy using a code buffer can be seen in Table 5.4. The columns represent the same data as in Section 5.4.

Notably, the performance degradation in the program `PrimeNumbers.Generate` is ameliorated slightly as the parameter for the number of primes to generate increases. This is in large part due to slightly more time being spent in the phase of the program dedicated to generating primes, rather than the final (and longest) phase of the program, where they are appended to the system log. However, the performance worsens slightly once more in the case of generating 150 primes; this is entirely due to the phase wherein primes are appended to the system log, as it has to print additional lines of primes, which are growing larger for every increase of the parameter. As noted in the previous section, this procedure is rather expensive due to a high level of interference. While its expense is certainly not as extreme as in the case where the entire **TextFrames** module has to be loaded repeatedly, it still requires repeatedly loading procedures in and out of the code buffer.

Additionally, an interesting difference in the results can be noted between

`PrimeNumbers.Generate` and `Hilbert.Draw`. While the former does not require loading too many procedures, it requires loading a considerable amount in terms of total code size, thus causing considerable performance degradation. On the other hand, the performance of `Hilbert.Draw` is worsened by loading an extreme amount of small procedures, thus spending a lot of time in overhead. Additionally, note that as mentioned in Section 3.4, at minimum a block of 512B must be loaded from secondary storage. As the recursive functions in Hilbert are considerably smaller than 512B, this leads to wasting time waiting for the block being read from disk to finish. The effect this has on performance has been seen in Section 5.1, and is echoed here.

5.5.2 System heap results

This section will present and discuss results from the final version developed in this thesis, presented in Section 3.6.2. To summarise, in this strategy overlays are allocated on the system heap and deallocated by using garbage collection, in which the dynamic construction of an overlay tree determines which overlays are to be collected. Additionally, as this strategy exploits the Table of Contents, the two approaches to position-independence presented in Section 3.5 will also be contrasted. Table 5.5 presents the results of running `PrimeNumbers.Generate` and `Hilbert.Draw` without filling the Table of Contents, and Table 5.6 presents these results with the Table of Contents. The columns represent the same thing as in Section 5.4 and Section 5.5. Finally, Table 5.7 notes the heap sizes used for the strained heap setups — the slight differences in heap size required to successfully run different testbenches with and without the Table of Contents comes down to multiple factors, in particular how differences in code size cause garbage collection to run multiple times, causing different levels of heap fragmentation.

As can be seen in Table 5.5, the amount of data that needs to be loaded into memory does not increase for the `PrimeNumbers.Generate` program regardless of its parameter, as the amount of code required to be in memory to run it remains the same. Thus, as soon as it has been loaded into memory once, it can remain there until it finishes executing. For the case of the strained heap, this largely holds, except that a few procedures are evicted and need to be loaded anew. Furthermore, the difference in performance between running programs on a strained heap and a heap that can fit the entire program (i.e. **not warmed up**) is fairly small, especially as the parameter for `PrimeNumbers.Generate` increases. This is in large part because the amount of code that has to be loaded anew on the strained heap is not considerable, across all tested programs, and the performance impact of garbage collection is fairly negligible. As can be seen, when Primes.Generate runs with a strained heap, only six procedures have to be loaded anew; this is due to the fact that the program runs in phases. When garbage collection occurs after the second phase, when the text is due to be appended to the system log, the only procedures that are used in the previous phases and in the last phase are in the **Texts** module, related to handling text pieces. A very similar situation occurs in `Hilbert.Draw`,

Table 5.5: Results of experiments run with a heap-based strategy, using the overlay manager for position-independent code.

Testbench	Overlay loads	Overlaid code (B)	Additional stack use (B)	Instructions run (overlays)	% increase (instructions)
No Table of Contents, strained heap					
Primes.Generate 25	38	26 112	208	375 268	425,8
Primes.Generate 50	38	26 112	208	487 114	212,2
Primes.Generate 75	38	26 112	208	601 460	147,0
Primes.Generate 100	38	26 112	208	718 145	116,0
Primes.Generate 125	38	26 112	208	836 940	90,74
Primes.Generate 150	38	26 112	208	981 561	90,76
Hilbert.Draw	56	34 560	196	2 961 518	83,20
No Table of Contents, not warmed up					
Primes.Generate 25	32	21 888	208	326 491	357,5
Primes.Generate 50	32	21 888	208	440 785	182,5
Primes.Generate 75	32	21 888	208	557 540	128,9
Primes.Generate 100	32	21 888	208	676 727	103,5
Primes.Generate 125	32	21 888	208	798 046	81,87
Primes.Generate 150	32	21 888	208	963 520	87,25
Hilbert.Draw	52	31 120	196	2 909 121	79,96
No Table of Contents, warmed up					
Primes.Generate 25	0	0	120	99 517	39,44
Primes.Generate 50	0	0	120	213 572	36,89
Primes.Generate 75	0	0	120	375 436	54,16
Primes.Generate 100	0	0	120	494 243	48,62
Primes.Generate 125	0	0	120	615 010	40,16
Primes.Generate 150	0	0	120	689 306	33,96
Hilbert.Draw	0	0	108	2 597 408	60,68

where garbage collection runs after the viewer has been created, and most of the evicted procedures are not reused, aside from for instance a drawing procedure in module **Display**.

In addition, the pattern of loading more overlays but less data that was seen in both Section 5.4 and Section 5.5 does not continue when they do not have to be loaded repeatedly. `Hilbert.Draw` uses more procedures *and* more memory in total; the difference is that it consists of calling many small procedures very many times, as was seen in Section 5.4 and Section 5.5. When these small procedures can stay in memory, performance improves drastically.

Finally, note that the results for a **warmed up** system do not require loading any code; as it has already been warmed up, the code has already been loaded into memory. Thus, these results represent the total overhead of the system when code has already been loaded into memory, i.e. when it just has to go through the overlay manager for each procedure call. The amount of data it places on the stack is lower than for the setups that also load code, but still considerable. This shows two things: firstly, loading code requires an additional $208\text{B} - 120\text{B} = 88\text{B}$ of data on the stack; this will always happen in the bottom of the call stack for an overlaid program. Secondly, there is considerable performance degradation in calling all procedures via the overlay manager; this is at its worst in `Hilbert.Draw` which additionally comprises many indirect function calls.

With regard to the strategy using the Table of Contents, a rather surprising result in Table 5.6 is the small additional stack size required to run the Hilbert testbench when it has been warmed up. As the program involves multiple layers of indirect function calls, one would expect a considerable impact on the total stack size required, as each indirect function call has to go through the overlay manager, which adds 12B on the stack. However, the reason for the low additional requirement in terms of stack size is that the stack does not grow to its largest size in the stage of the program involving recursion, i.e. when the Hilbert curve is drawn, but in the program's first phase, where the viewer on the left side of the screen is created. This phase only involves a single indirect procedure call, and as such, this setup only stresses the stack by an additional 12B. A similar result can be seen for `PrimeNumbers.Generate`, which requires two indirect procedure calls to draw text.

Notably, some additional code has to be loaded into memory to support the Table of Contents strategy. This is due to the additional overhead involved in supporting the strategy, as the mark is now decremented/incremented in the overlaid code, as explained in Section 3.9.2, rather than only in the overlay manager. This increased amount of code that has to be transferred from secondary storage is made up for by the increase in performance this allows, particularly for programs that run for a longer period of time.

Finally, the only procedure that has to be loaded into memory twice in Hilbert using a Table of Contents is `Display.ReplConst`. Notably, the version that does not exploit the Table of Contents evicts more than the one that does, despite the latter having a larger code size. They both run garbage collection once to evict overlays,

Table 5.6: Results on experiments run with a heap-based strategy, using the Table of Contents for position-independent code.

Testbench	Overlay loads	Overlaid code (B)	Additional stack use (B)	Instructions run (overlays)	% increase (instructions)
Table of Contents, strained heap					
Primes.Generate 25	38	30 464	208	391 978	449,2
Primes.Generate 50	38	30 464	208	485 971	211,5
Primes.Generate 75	38	30 464	208	584 224	139,9
Primes.Generate 100	38	30 464	208	681 263	104,9
Primes.Generate 125	38	30 464	208	782 024	78,22
Primes.Generate 150	38	30 464	208	909 140	76,69
Hilbert.Draw	53	35 968	196	2 550 319	57,77
Table of Contents, not warmed up					
Primes.Generate 25	32	25 856	208	318 564	346,3
Primes.Generate 50	32	25 856	208	412 544	164,4
Primes.Generate 75	32	25 856	208	508 985	109,0
Primes.Generate 100	32	25 856	208	607 812	82,77
Primes.Generate 125	32	25 856	208	708 569	61,48
Primes.Generate 150	32	25 856	208	829 522	61,21
Hilbert.Draw	52	34 944	196	2 505 894	55,02
Table of Contents, warmed up					
Primes.Generate 25	0	0	24	80 005	12,10
Primes.Generate 50	0	0	24	173 630	11,29
Primes.Generate 75	0	0	24	291 078	19,52
Primes.Generate 100	0	0	24	389 417	17,10
Primes.Generate 125	0	0	24	489 622	11,58
Primes.Generate 150	0	0	24	567 905	10,37
Hilbert.Draw	0	0	12	2 174 178	34,50

Table 5.7: The minimal heap used for the *strained heap* testbenches.

Testbench	Minimal heap
No Table of Contents	
Primes.Generate	49 920
Hilbert.Draw	49 152
Table of Contents	
Primes.Generate	53 504
Hilbert.Draw	53 504

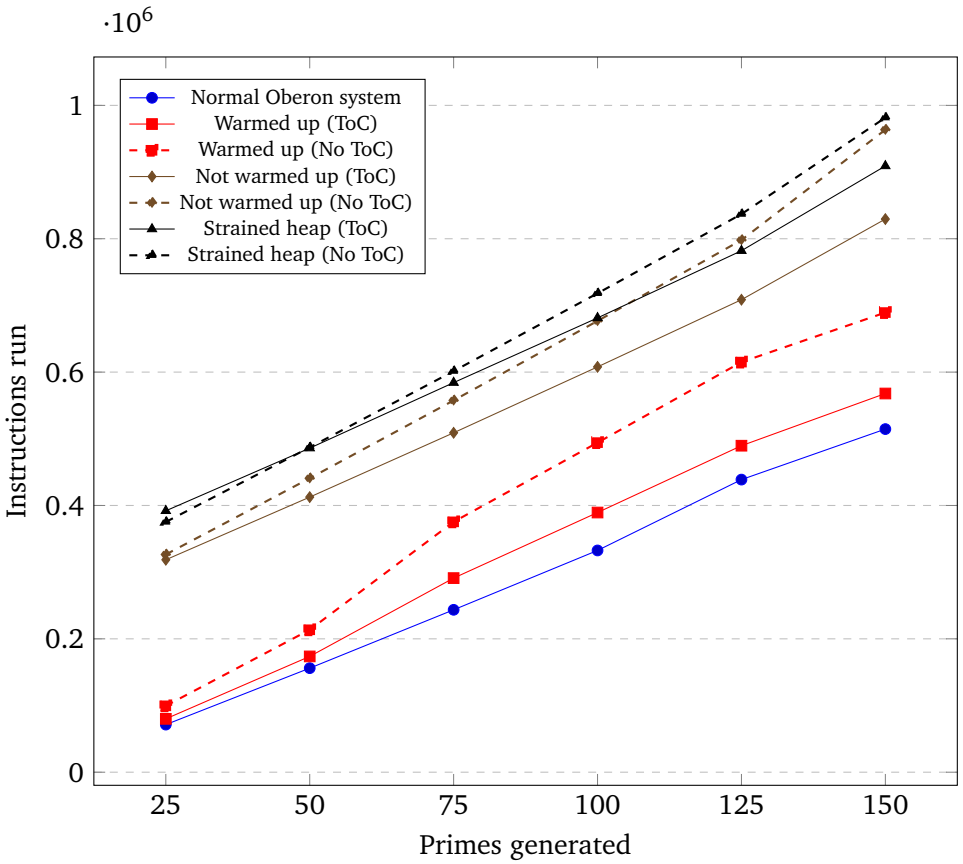


Figure 5.2: The performance of PrimeNumbers.Generate given different setups, using an overlay tree strategy.

but the version using a Table of Contents needs to do so slightly later than the version that does not. Specifically, it runs *after* the viewer has been initialised, and therefore evicts more code that doesn't need to be loaded anew. This is due to the version using a Table of Contents requiring a larger heap to successfully boot, link and run the program (see Table 5.7), but the program itself strains the heap slightly less.

The differences in performance of PrimeNumbers given different setups and a different number of generated primes are also given in Figure 5.2, to fully capture the different relationships between heap size, position independence strategy, and performance.

The differing performance between a warmed up setup using the Table of Contents and a standard Oberon system shows that the impact of the overhead incurred by the indirection of the Table of Contents as well as additional code for supporting a code overlay strategy is rather minimal, but not insignificant. In total it averages a 13.66% performance degradation. The vast majority of this degradation is due to procedure calls, which under this scheme must both fetch

Table 5.8: Minimum amount of memory required to boot. The *Overlays* column contains information on overhead used by the overlay system, i.e. the overlay table and overlay region. Note that memory required to include the bitmapped display, 98304B, is not included in the total. Sizes are given in bytes.

Setup	Stack	Static code/data	Heap	Overlays	**Total**
Standard Oberon system	10 188	130 944	37 376	0	178 508
Overlay tree with ToC	10 208	54 788	48 896	3 168	117 060
Overlay tree without ToC	10 208	54 436	47 360	3 168	115 172
Function-granularity with code buffer	10 208	52 624	27 440	5 840	96 112
Module-granularity with code buffer	10 208	52 620	57 344	35 068	155 240

the location of the code segment to jump to, decrement the mark, and increment the mark upon returning, as discussed in Section 3.9.2.

However, while this does represent some amount of performance degradation, using the Table of Contents still allows significantly better performance than not doing so for every setup. This is in large part due to the fact that using the Table of Contents allows the program to skip the overlay manager, instead jumping directly to the overlaid procedure in question. Thus, for any program that uses the same procedures over and over, it can achieve considerable speedups. It is even quite pronounced for Hilbert, which, despite the Table of Contents not aiding the indirect procedure calls, can now more efficiently call the procedures related to creating the viewer and drawing the lines of the curve itself. This is especially notable when comparing it to a warmed up setup without a filled out Table of Contents. As can be seen in Figure 5.2, while the warmed up setup using the Table of Contents is able to achieve acceptable performance, the setup without the Table of Contents sees significant performance degradation as the parameter is increased, in large part due to having to display more lines of text, as has already been discussed.

5.6 Booting

This chapter will end with a discussion on booting the system under the different configurations presented in this thesis.

The performance in terms of instructions run while booting the Oberon system under different configurations can be seen in Figure 5.3. Note that these results include the time it takes to link and create overlays, because, as explained in Section 3.3, this is done in the Oberon runtime. Furthermore, the minimal amount of memory required to fully boot the system is given in Table 5.8. (Due to an

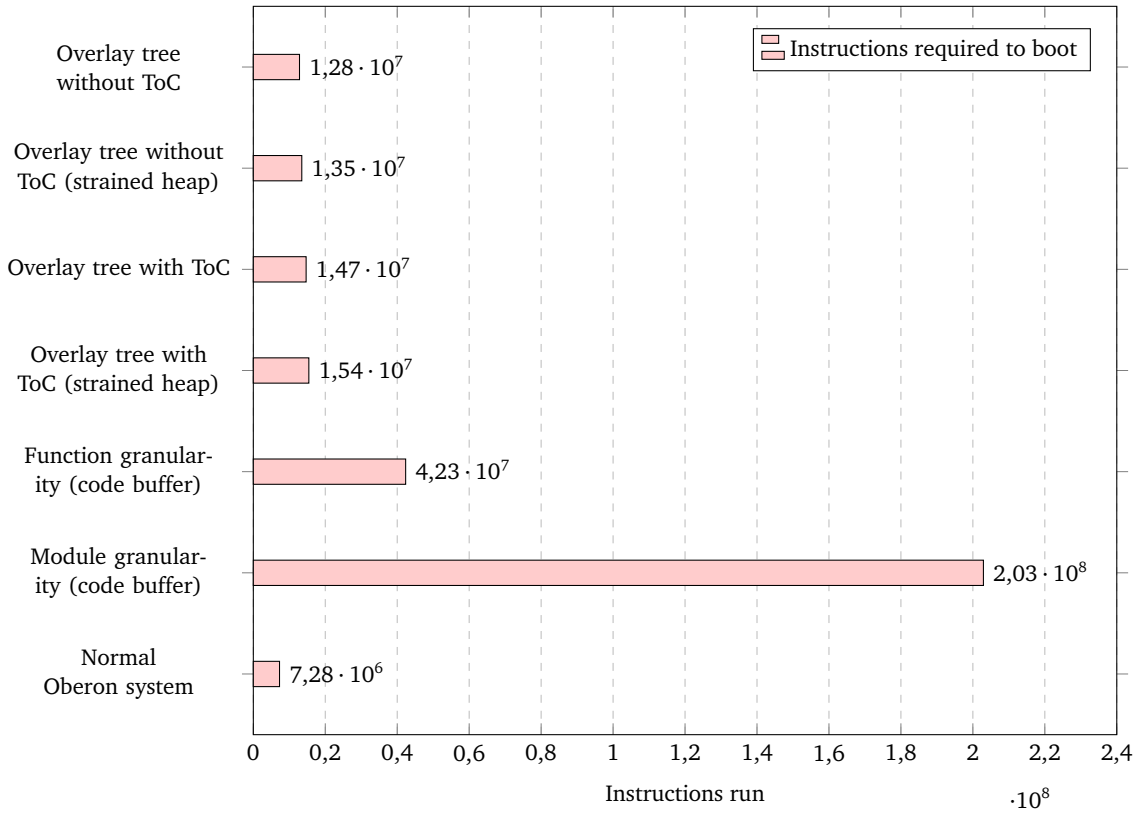


Figure 5.3: Instructions required to boot a full Oberon system, given the different strategies presented in this thesis, as well as whether the heap is strained. ToC is short for Table of Contents.

implementation detail of the overlay system, the amount of space allocated on the heap to contain the sector map has been halved, reducing its size by 4kB. To make the results a fair comparison, this has also been done in the standard Oberon system; no other changes were made.)

While these strategies and their results with regard to booting will be compared in more depth in Section 6.3, some aspects will be briefly highlighted here.

As has been seen in other experiments, module-granularity overlays also had the worst performance in booting the system, without having a significant improvement in memory footprint. This performance is, as in the other testbenches, largely due to having to copy large modules into memory over and over. While the memory footprint of this strategy could likely be improved somewhat, this strategy only served as a stepping stone in this thesis, and as such we saw no need to further develop it.

Another thing of note is the extreme strain on the stack across every setup. This is unrelated to overlay strategies, whose actual effect on the stack have been highlighted elsewhere: in the procedure `FileDir.Init`, responsible for initialising the file system, an array of 2000 integers is placed on the stack (totalling 8000B), along with an additional 2000B of additional overhead to keep track of which sectors need to be marked. Improving the memory footprint of these procedures is outside the scope of this thesis, but a solution will quickly be noted: considering the discussion of the improved garbage collection scheme in Section 5.3, there would be no issue with placing most of this on the heap rather than the stack, to let it be garbage collected for when the heap is needed later. This would have a minimal impact on the heap, as no other data is needed before this block of 10kB can be freed[2].

An interesting aspect seen in these results is that booting with the Table of Contents is actually slower than the experiment detailed in Section 5.5. This is due to the fact that the booting process consists in large part of calling many procedures once, rather than the same procedure repeatedly; thus, the performance increase offered offered in previous experiments is offset by the increased overhead of filling in the Table of Contents for each loaded overlay.

Finally, while Table 5.8 lists *minimal* configurations that are required to boot the system, this by no means indicate that they produce a functioning Oberon system. The versions on the heap, in particular, are able to successfully reach the main Oberon loop with the given setup; however, due to the small heap, this is with such extreme heap fragmentation that many aspects of the system do not work due to being unable to allocate a block large enough to hold a code overlay. The impact of heap fragmentation will be discussed in more detail in Section 6.2.

[2]In addition, a potential way to avoid any fragmentation at all is to allocate the rest of the heap for this purpose, and let garbage collection deallocate it after the procedure is completed; this would return the heap to its previous state. This is not, however, an elegant solution.

Chapter 6

Evaluation and Future Work

This chapter will evaluate the project as a whole, as well as suggest future work that builds on the contributions of this thesis. First, specific limitations in the project with regard to the emulated hardware platform and its secondary storage are discussed in Section 6.1. Then, the issue of heap fragmentation is briefly addressed in Section 6.2, before the evaluation concludes with a consideration of the Pareto optimality of the strategies developed in this thesis in Section 6.3. This is followed by Section 6.4, which gives some ideas for potential future work.

6.1 Limitations of overlays using SD cards

In section 3.4, a method for loading code segments from secondary storage into memory directly rather than via the use of the file system was presented. The differing results in performance between the two methods were presented in Section 5.1. There are, however, still further optimisations that can be done for reading/writing data from/to the SD card as efficiently as possible.

Firstly, it would be ideal to only read/write the number of bytes in the block that are currently needed. In an SD card, this is known as partial block reading/writing, where the SD card is given information on what segment of the block is needed, and only transfers that rather than the entire block. While using this feature would lead to a slight improvement in performance, partial block reading/writing is disallowed by the SD Card specification for SDHC and SDXC cards, and is also not supported by all regular SD cards [25]. Therefore, the current solution of reading entire blocks and ignoring data past what is necessary is as efficient as it can be for most SD cards.

Secondly, only single block read/write was used, but for reading/writing large code segments, using multiple block read/write would likely lead to increased throughput [25]. However, the emulator in use did not support those SD card commands, and it was therefore not implemented. The result of using multiple block read/write rather than single block would regardless not be representative of performance on real hardware.

Finally, as noted in Section 4.2.1, the performance of an SD card is highly variable based on the specific SD card chosen. Additionally, it must be acknowledged that as secondary storage, SD cards are generally poorly suited for the goal of this thesis [30]. Although the thesis has assumed usage of the same storage medium as the original Oberon system, the approach is not in any way incompatible with other storage mediums, and in fact would be better suited for them. There are only a few specifics in the source code that would have to be changed:

- **The mechanism for choosing where overlays are stored on disk**: Currently, the location of each overlay is indicated by the sector of the SD card they have been written to. If one chooses to use non-volatile memory, for instance, this could be changed to the NVM address at which the code segment is stored.
- **The SD card driver**: Both the bootloader and the kernel use an SD card driver to read and write data from the disk over SPI. This driver would have to be replaced.

As such, although the current implementation could be further improved to better improve throughput from SD cards, the better choice upon implementing it in hardware is likely to use a different secondary storage altogether.

6.2 Heap fragmentation

Over time, the heap-based strategy will gradually lead to memory fragmentation. This is due to two unfortunate characteristics of the strategy, combined with an ill-suited heap allocator and garbage collector.

Firstly, as noted in Section 2.2.4, the garbage collector only merges free blocks that are collected simultaneously. If two adjacent blocks are allocated, and they are freed in separate iterations of garbage collection, they will not be merged, but rather become two adjacent, fragmented free blocks. This is noted as a major cause of fragmentation in [14] (in which it is termed "isolated deaths").

Secondly, the system experiences much more frequent, varying use of the heap allocator when placing code overlays on the heap. Far more blocks are allocated on the heap (as evidenced by garbage collection running ten times to allow a strained system to boot, as noted in Figure 5.3), and those blocks are of varying size, as procedures' sizes vary considerably in the Oberon system. If a program that allocates many small code segments on the heap, stressing the heap such that garbage collection must run several times, and then needs to allocate a large code segment, the many small allocations may have fragmented the heap enough to make this impossible. This, too, is noted as a cause of fragmentation in [14].

Although this is certainly an issue, it is by far most notable when the heap is particularly strained. Running with the minimal setup for booting on the heap with the Table of Contents given in Section 5.6, the largest free block on the heap after booting has finished is of size 2304B — not necessarily small, but too small to fit the largest procedure in the system, which would cause a problem if it needed

to be loaded into memory. However, increasing the size of the heap by 8448B (to a total size of 58368B) requires garbage collection to run less frequently, and as such it is able to merge more adjacent unmarked blocks when it does run. This leads to the largest block on the heap being 7680B, which is more than large enough to fit the largest procedure in the system.

However, while increasing the size of the heap ameliorates the issue by addressing the first cause of heap fragmentation, it does not address the second, nor does it ensure that the heap will not fragment eventually, either due to a program that stresses the heap or simply due to use over time. While this issue has not been solved in this thesis, there are many potential solutions.

One solution would be to create an additional "code buffer" within the heap. This would be a block that is set to never split, exclusively for use by the overlay system. It would have two use-cases: firstly, to be used for particularly large code segments, to avoid having to allocate them on the heap; and secondly, to ensure that the system can continue running even if the heap fragments. This would essentially allow the overlay system to change strategy to the function-granularity code buffer strategy presented in Section 3.9.1, if the heap is fragmented enough that it cannot be used for most code overlays anymore.

This would not solve the fundamental problem of a heap fragmenting. As the problem lies largely with Oberon's heap system, the most beneficial solution would be to create a heap allocator and garbage collector that are designed with this use-case in mind, and as such can handle fragmentation more gracefully. However, that falls outside the scope of this thesis.

6.3 Pareto optimality

The Pareto optimality (see section 2.4) of the different strategies presented in this thesis can be evaluated by establishing two objective functions. As the focus of the thesis is on overlaying the Oberon system itself, the analysis will restrict itself to the case of booting the system[1]. Two objective functions are chosen for which to optimise: instructions run to fully boot the system, and total memory footprint.

An approximate Pareto front is established in Figure 6.1, by comparing the different strategies explored in this thesis in terms of instructions run to boot the system and memory footprint. While module-granularity overlays were also considered in this thesis, that strategy was not included, as despite some decrease in memory footprint, its poor performance would render the rest of the figure illegible.

As is evident, none of the strategies proposed in this thesis are more performant than the original Oberon system; therefore, among the solutions evaluated, it is approximately Pareto optimal, as none of the solutions presented in this thesis lower the memory footprint without also impacting performance. As explained

[1]While this analysis has been done in the case of booting, the resulting evaluation will necessarily differ if the objective function is changed to measure the performance of a different program.

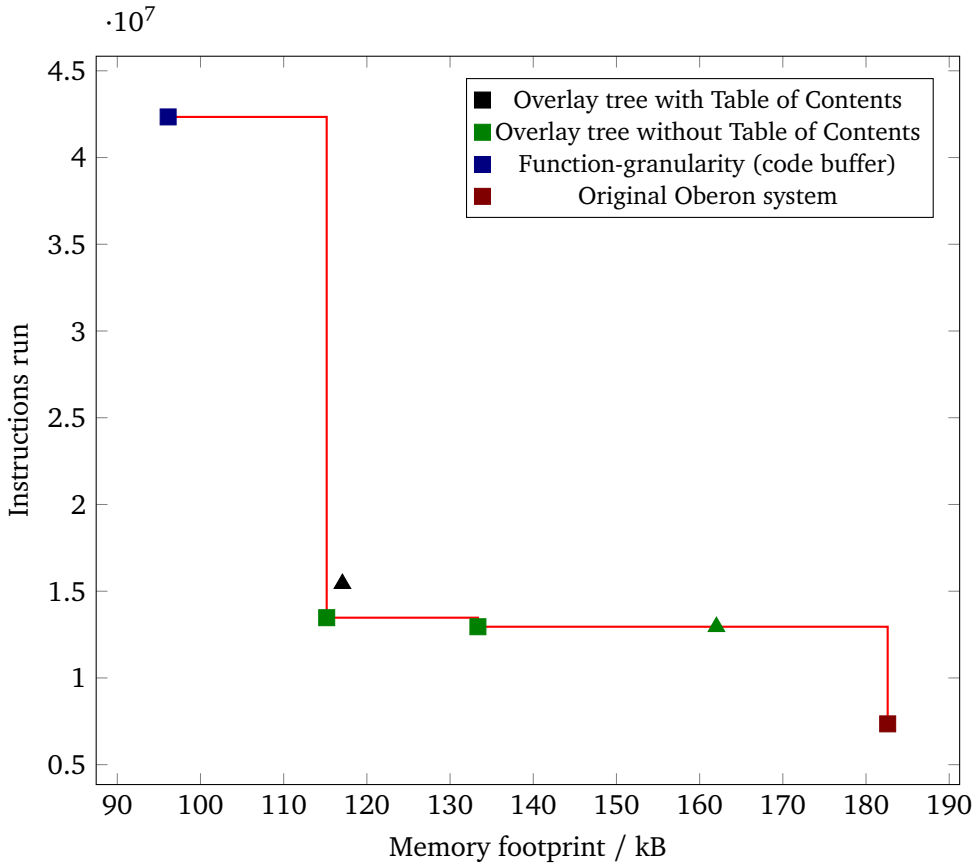


Figure 6.1: Depiction of the Pareto front approximations of the different strategies evaluated in this thesis, in the case of booting the Oberon system. Only solutions with a strained heap are included for the heap-based strategy. Squares indicate approximate Pareto optimal solutions, and triangles indicate dominated solutions.

in Section 2.4, this does not mean it is Pareto optimal: a solution with the same memory footprint as the original Oberon system but slightly better performance, for instance, almost certainly exists. However, from the solutions presented in this thesis, it will still be a good alternative for systems of certain configurations that can support its memory footprint.

The other approximate Pareto optimal extreme presented in this thesis is using function-granularity with a single code buffer. This solution is able to achieve the lowest possible memory footprint, by overlaying every single procedure into a single code buffer. It does this at a rather steep cost in performance, but it is also the only solution able to achieve a memory footprint below 100kB (excluding the frame buffer), which may be the only solution that satisfies the hardware requirements of some platforms.

The solutions placing code overlays on the heap are able to achieve a more tolerable compromise between performance and memory footprint. First of all, note that including the overhead of updating the Table of Contents actually leads to a solution that isn't approximately Pareto optimal in the case of booting the system. Despite achieving considerably better results in the other testbenches, booting the Oberon system consists in large part of running procedures once, to initialise different structures. Thus, the performance gained by using the Table of Contents is lost to the overhead of updating it.

While the solutions using a code buffer — both for module- and function-granularity overlays — cannot tweak the size of the code buffer (as it is bounded by the maximum sizes of modules and functions respectively), the solutions utilising the heap can. Thus, for the overlay tree strategy, the size of the heap can be increased for additional performance, as that would not require garbage collection to run as often. However, evidently, this has a low impact on performance for booting the system. Three different total memory footprints have been plotted, according to increases in heap size; as can be seen, increasing the size of the heap by 5kB allows for a slight performance increase, as garbage collection no longer has to run while the system is booted. However, increasing the size of the heap further, by another 30kB, causes no additional improvement in performance at all; thus, this solution is dominated and not approximately Pareto optimal.

In summary, the different strategies presented in this thesis have different use-cases depending on the system in question, and its requirements with regard to the trade-off between performance and memory footprint. It cannot be claimed that any solution presented is better than the original Oberon system for all use cases, as performance degradation caused by moving code from secondary storage to memory is unavoidable.

6.4 Future work

In this section, ideas for future work that builds on the contributions of this thesis will be presented.

6.4.1 Testing on hardware

Currently, the solution is only tested in a RISC-V emulator; ideally it should also be tested on real hardware. The simplest way to do this would be to use the same FPGA board as the Project Oberon 2013 was designed for, i.e. the Xilinx UG130 Spartan Starter Kit board [2], and flash a RISC-V processor onto it. This would naturally have to be augmented with circuitry supporting PS/2 ports, the VGA port, etc. Additionally, as the board in question is quite old, first released in 2004 [32], it may be somewhat difficult to acquire, though it should certainly still be possible.

6.4.2 A dynamic Least Recently Used strategy on the heap

As part of this thesis, an implementation of a least recently used (LRU) strategy was started. The core idea was to extend the heap metadata prefix of overlays by one word, on which a timestamp could be placed. Timestamping can be done in a variety of ways, though the one most easily supported by the RISC-V specification is to retrieve a time or cycle count from control status registers (CSRs) dedicated to those purposes, which the specification requires compliant RISC-V implementations to support [9].

This could also be integrated into garbage collection, by communicating with the garbage collector how much data is needed; the garbage collector could then evict the least recently used overlays by checking their timestamps, such that enough data is freed for the block about to be allocated.

This strategy could potentially be useful. It would require some additional overhead, but it would also solve some of the issues in the overlay tree approach, such as overlays far up in the tree that won't be needed for quite some time taking up space in the heap.

6.4.3 Sophisticated overlay strategies

While the strategy for dynamically creating an overlay tree on the heap works reasonably well, the overhead of loading overlays into memory causes a significant hit to performance. A way to lower this overhead would be to pursue more complex strategies of grouping functions into larger overlays, as is commonly done [7][6]. A result that strongly motivates performing such an improvement was seen in section 5.5, in particular the results for when the system has already been warmed up. By making the warmed up version into a single overlay — or perhaps three, for the three different phases of the PrimeNumbers program, as mentioned in section 4.3.4 — much better performance can be achieved. As is well-known, and discussed in closer detail in section 6.1, reading larger segments linearly from secondary storage is more performant than random reads, which would also be a benefit of this suggested improvement.

However, there are a couple of issues that would have to be solved for this approach to be feasible. Firstly, as noted in section 3.3, overlays are currently

generated in the Oberon runtime; applying more complex strategies, e.g. by analysing the call graph [5][7], would likely have a significant impact on the time it takes to generate overlays, which is already quite high. Secondly, this would not only have an impact on performance, but also on code size — and as this would, under the current system, have to be done in the inner core, the code for generating more complex overlays would have to remain resident in memory. (A potential solution to the latter problem is presented in section 6.4.5.)

6.4.4 Identifying hot code

A strategy used in [7], which explores overlaying the Linux kernel, has the potential to also be quite efficient for the Oberon system. The strategy is to, in addition to code that has to always be kept in memory, allow a certain amount of additional, frequently accessed code to be memory-resident; in the paper in question, allowing only 4% of additional code (or 10kB in their case) to be memory-resident leads to a performance increase of one to two orders of magnitude.

This could lead to a dramatic increase in performance, particularly if a code buffer is used, as less code that is used very frequently by the Oberon system would have to be overlaid. For instance, many of the procedures in the **Display** module are frequently used, as any procedure that involves either printing text or displaying anything on the screen will have to use them; these could be made memory-resident to avoid repeatedly having to read them from secondary storage. However, implementing such a system in Oberon is not without its obstacles. It would require correctly identifying which procedures to make memory-resident, which in other words requires knowing which procedures will be needed frequently. No runtime information is currently gathered beforehand, so this would have to be done either with static analysis (identifying procedures called by many other procedures), or by monitoring the system over time to identify procedures that are frequently required.

6.4.5 Towards an Oberon Microkernel

In this project, potential ways of optimising the Oberon system's memory footprint have been explored. While the Oberon system was already quite small, this project has succeeded in making it smaller. In this section, additional ways of making it even smaller, in the trajectory of an Oberon microkernel, are presented as an idea of which direction this project can be used for next. The Oberon system was incorporated into a microkernel in [33], though not with the goal of overlaying the operating system in mind, but rather to allow it to coexist on the same system with other operating systems on the same machine using a general-purpose microkernel. The solution proposed here is a special-purpose microkernel, built for the requirements of the Oberon system.

The Oberon kernel consists of a heap allocator, garbage collection procedures, a disk driver, and a trap handler. It is already quite small: the RISC-5 Oberon kernel occupies only 1123 words of code. However, with the contributions made

in this thesis, it can be made much more minimal, particularly when the hardware requirements of the Oberon system are considered. In fact, the only procedures that truly must always be resident in memory are the overlay manager and the trap handler, as other parts of code must be able to jump to these at any time. Everything else may be overlaid as needed, including garbage collection as well as the mechanisms for linking code and generating overlays. This would result in a very small kernel. Unfortunately, as the overlay manager must be able to load code from secondary storage into main memory, a basic driver for interacting with the disk would also need to be resident in memory.

Currently, the system generates overlays in runtime, and the inner core (i.e. the kernel, file system, overlay system, and linker) is always resident in memory. To allow for the system described here to work, these would also have to be overlaid before booting the system. While this may seem like a large deviation from the Oberon system, consider that these must already be linked beforehand by a different host: it is not a stretch to also create overlays for them beforehand.

Naturally, it is debatable whether this would be a true microkernel. Liedtke notes in [34] that a microkernel should only implement that which cannot be moved outside it; and while that would be true for the proposed iteration described here, many of the features deemed necessary for a microkernel in his paper would not be implemented, due to the Oberon system having no need for them. We argue that, particularly if the issue of the disk driver being in the kernel is resolved, this would still qualify as a microkernel, as it would fulfil truly the minimal amount required for a special-purpose Oberon kernel, by essentially only including a small memory manager.

This has not been done in this thesis, as it is outside of its scope, and it is a considerable undertaking. However, this thesis lays the groundwork for such an approach to work. Furthermore, it would certainly be in the spirit of the Oberon project to construct such a small, simple system; it would fulfil the goal of being "lean software", as explained in Section 2.2.1, constraining the memory-resident code of an entire OS to a few kilobytes of memory.

Chapter 7

Conclusion

The goal of this thesis was to explore whether code overlays were a tenable strategy for reducing the memory footprint of the Oberon system, a lean operating system [3], running on RISC-V. To that end, the main contribution of this thesis, a prototype with multiple different configurations that have different characteristics, was developed.

Among these, different trade-offs between performance and memory footprint were found, with one strategy being able to reduce the memory footprint of the Oberon system below 100KB if one does not include the display, though with a significant impact on performance, causing a 476% increase in instructions run upon booting the Oberon system. Another strategy, which allocates space on the system heap for code overlays, achieves better performance, though still with 84.9% performance degradation. These results do, however, include linking and generating overlays. On programs running on top of the Oberon system, better performance was observed, particularly in programs that run long enough to warm up.

To answer the research goal of the thesis, it would appear that code overlays *are* a tenable strategy for reducing the memory footprint of the Oberon system, though not without a loss in performance. There are also some obstacles unique to the system, such as the dynamic linker and loader; despite these obstacles, the Oberon system also allowed the implementation of a strategy that to our knowledge has not been attempted before, which is using garbage collection to evict overlays.

We have presented ideas to improve this performance — such as more sophisticated strategies for combining procedures into larger overlays, and making hot code in the outer core memory-resident — for future work that can further improve the performance while maintaining a decreased memory footprint.

Bibliography

[1] R. Solbjørg, 'Porting the Oberon system to the RISC-V instruction set architecture,' 2020. [Online]. Available: https://github.com/solbjorg/oberon-riscv/blob/master/report.pdf.

[2] N. Wirth and J. Gutknecht, *Project Oberon: The Design of an Operating System, a Compiler, and a Computer*. 2013.

[3] N. Wirth, 'A plea for lean software,' *Computer*, vol. 28, no. 2, pp. 64–68, 1995. [Online]. Available: https://people.inf.ethz.ch/wirth/Articles/LeanSoftware.pdf.

[4] *riscv/riscv-overlay*. [Online]. Available: https://github.com/riscv/riscv-overlay (visited on 15/02/2021).

[5] R. Cytron and P. Loewner, 'An automatic overlay generator,' 1986.

[6] C. Jang, J. Lee, B. Egger and S. Ryu, 'Automatic code overlay generation and partially redundant code fetch elimination,' *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 2, pp. 1–32, Jun. 2012, ISSN: 1544-3566, 1544-3973. DOI: 10.1145/2207222.2207226. [Online]. Available: https://dl.acm.org/doi/10.1145/2207222.2207226 (visited on 10/02/2021).

[7] H. He, S. K. Debray and G. R. Andrews, 'The revenge of the overlay: automatic compaction of OS kernel code via on-demand code loading,' in *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, 2007, pp. 75–83.

[8] D. Patterson and A. Waterman, *The RISC-V Reader: an open architecture Atlas*. Strawberry Canyon, 2017.

[9] A. Waterman and K. Asanovic, *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*, 2019.

[10] N. Wirth. (2016). 'The Programming Language Oberon,' [Online]. Available: https://people.inf.ethz.ch/wirth/Oberon/Oberon07.Report.pdf.

[11] N. Wirth, *The RISC Architecture*, 2018.

[12] M. Franz, 'Oberon - The Overlooked Jewel,' in *The School of Niklaus Wirth*, 2000, pp. 41–54.

[13] C. Yang, 'Linkers and Loaders,' p. 299, 1999.

[14] P. R. Wilson, M. S. Johnstone, M. Neely and D. Boles, 'Dynamic storage allocation: A survey and critical review,' in *International Workshop on Memory Management*, Springer, 1995, pp. 1–116.

[15] Y. Kim, J. Cai, Y. Kim, Kyoungwoo Lee and A. Shrivastava, 'Splitting functions in code management on scratchpad memories,' in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, ISSN: 1558-2434, Nov. 2016, pp. 1–8.

[16] P. Naur, 'The performance of a system for automatic segmentation of programs within an ALGOL compiler (GIER ALGOL),' *Communications of the ACM*, vol. 8, no. 11, pp. 671–676, Nov. 1965, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/365660.365680. [Online]. Available: https://dl.acm.org/doi/10.1145/365660.365680 (visited on 22/03/2021).

[17] R. J. Pankhurst, 'Operating Systems: Program overlay techniques,' *Communications of the ACM*, vol. 11, no. 2, pp. 119–125, Feb. 1968, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/362896.362923. [Online]. Available: https://dl.acm.org/doi/10.1145/362896.362923 (visited on 22/03/2021).

[18] W. Gay, 'Code Overlays,' in *Beginning STM32: Developing with FreeRTOS, libopencm3 and GCC*, W. Gay, Ed., Berkeley, CA: Apress, 2018, pp. 147–174, ISBN: 978-1-4842-3624-6. DOI: 10.1007/978-1-4842-3624-6_9. [Online]. Available: https://doi.org/10.1007/978-1-4842-3624-6_9 (visited on 22/01/2021).

[19] P. Lokuciejewski, S. Plazar, H. Falk, P. Marwedel and L. Thiele, 'Multi-objective exploration of compiler optimizations for real-time systems,' in *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, IEEE, 2010, pp. 115–122.

[20] D. A. Van Veldhuizen, G. B. Lamont *et al.*, 'Evolutionary Computation and Convergence to a Pareto Front,' in *Late breaking papers at the genetic programming 1998 conference*, Citeseer, 1998, pp. 221–228.

[21] R. Haen, O. Shinaar and C. Blackmore, *Software Overlay for RISCV-HLD*, 2021. [Online]. Available: https://github.com/riscv/riscv-overlay/blob/69f30f1093bc0a91a5a620f5cefca1e12083d36f/docs/overlay-hld.pdf.

[22] *RISC-V ELF psABI specification*. [Online]. Available: https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md (visited on 23/05/2021).

[23] A. S. Waterman, 'Design of the RISC-V instruction set architecture,' Ph.D. dissertation, UC Berkeley, 2016.

[24] C. Silvers, 'UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD,' in *2000 USENIX Annual Technical Conference (USENIX ATC 00)*, San Diego, CA: USENIX Association, Jun. 2000. [Online]. Available: https://www.usenix.org/conference/2000-usenix-annual-technical-conference/ubc-efficient-unified-io-and-memory-caching.

[25] SD Card Association, *SD Specifications Part 1: Physical Layer Simplified Specification, Version 8.00*, 2020.

[26] *An overview of the TOC on AIX*, Nov. 2012. [Online]. Available: http://www.ibm.com/developerworks/rational/library/overview-toc-aix/index.html (visited on 21/01/2021).

[27] Apple Computer, Inc., 'Inside Macintosh: Memory,' 1992.

[28] R. Shahriyar, S. M. Blackburn and K. S. McKinley, 'Fast conservative garbage collection,' in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, Portland Oregon USA: ACM, Oct. 2014, pp. 121–139, ISBN: 978-1-4503-2585-1. DOI: 10.1145/2660193.2660198. [Online]. Available: https://dl.acm.org/doi/10.1145/2660193.2660198 (visited on 17/05/2021).

[29] H.-J. Boehm and M. Weiser, 'Garbage collection in an uncooperative environment,' *Software: Practice and Experience*, vol. 18, no. 9, pp. 807–820, 1988.

[30] H. Kim, N. Agrawal and C. Ungureanu, 'Revisiting storage for smartphones,' *ACM Transactions on Storage (TOS)*, vol. 8, no. 4, pp. 1–25, 2012.

[31] N. Wirth, 'The Design of a RISC Architecture and its Implementation with an FPGA,' ETH Zurich, Department of Computer Science, Tech. Rep., 2015. [Online]. Available: https://people.inf.ethz.ch/wirth/FPGA-relatedWork/RISC.pdf.

[32] Xilinx Inc, *Spartan-3 FPGA Starter Kit Board User Guide*, 2008.

[33] J. A. De Villiers, 'Micro-kernel support for a lightweight extensible workstation operating system,' M.S. thesis, University of Stellenbosch, 1999.

[34] J. Liedtke, 'On micro-kernel construction,' *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5, pp. 237–250, 1995.

Appendix A

Test programs and data

A list of test programs as well as data used to gather results follows below.

A.1 Data transfer from disk to memory

Files used in testing data transfer from disk to memory follow below. Lorem ipsum text was generated using `https://lipsum.com`.

File1.txt, size 512B:

```
Ut eu lectus a tortor sollicitudin cursus. Suspendisse consectetur ante mattis
    ↪ elementum consequat. Duis mauris purus, porta vel sapien ac, commodo
    ↪ suscipit mi. Phasellus viverra, lacus euismod hendrerit consequat, lectus
    ↪ felis imperdiet neque, vitae cursus leo lectus nec turpis. Aenean in
    ↪ hendrerit odio. Donec sed orci eget mi laoreet tincidunt. Donec ultricies
    ↪ malesuada odio nec iaculis. Etiam finibus commodo libero volutpat euismod.
    ↪ Fusce efficitur augue pharetra quam convallis ullamcorper. Donec ligula
```

File2.txt, size 512B:

```
Donec elementum dolor id neque molestie vulputate laoreet non tellus. Nunc finibus,
    ↪  ex in aliquam pulvinar, augue ligula laoreet enim, in efficitur nulla orci
    ↪  ac nisl. Nullam pulvinar ultrices felis, sit amet tristique metus egestas
    ↪ sit amet. Nullam risus magna, maximus sed aliquam ut, vulputate sit amet
    ↪ tortor. Vestibulum hendrerit mauris eget mauris malesuada, a consectetur mi
    ↪  pretium. Mauris non nibh non tellus fringilla bibendum at eleifend velit.
    ↪ Donec hendrerit ipsum nunc, ac laoreet dui suscipit ut.
```

File3.txt, size 513B:

```
Ut eu lectus a tortor sollicitudin cursus. Suspendisse consectetur ante mattis
    ↪ elementum consequat. Duis mauris purus, porta vel sapien ac, commodo
    ↪ suscipit mi. Phasellus viverra, lacus euismod hendrerit consequat, lectus
    ↪ felis imperdiet neque, vitae cursus leo lectus nec turpis. Aenean in
    ↪ hendrerit odio. Donec sed orci eget mi laoreet tincidunt. Donec ultricies
    ↪ malesuada odio nec iaculis. Etiam finibus commodo libero volutpat euismod.
    ↪ Fusce efficitur augue pharetra quam convallis ullamcorper. Donec ligula.
```

File4.txt, size 1025B:

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam vehicula felis vel
    ↪ est lobortis accumsan. Proin rhoncus purus rutrum, efficitur ante quis,
    ↪ tincidunt velit. Morbi a fermentum sem. Donec sit amet augue tortor. Nam
    ↪ eget sagittis tortor. In hac habitasse platea dictumst. Pellentesque luctus
    ↪  metus magna, ac sagittis ligula varius quis. Aenean mi velit, volutpat ut
    ↪ justo in, luctus vulputate ligula. Donec ut mattis massa. Aliquam erat
    ↪ volutpat. Fusce ac turpis id elit suscipit accumsan. Integer rhoncus cursus
    ↪  placerat. Donec eget ultricies orci. Orci varius natoque penatibus et
    ↪ magnis dis parturient montes, nascetur ridiculus mus. Suspendisse interdum,
    ↪  mauris sed egestas gravida, urna ante maximus nulla, et eleifend libero
    ↪ lorem quis urna. Nullam facilisis nec magna eget facilisis. Proin sed
    ↪ laoreet risus. Proin aliquet facilisis nunc, vitae placerat risus facilisis
    ↪  id. Aliquam quis orci ex. Lorem ipsum dolor sit amet, consectetur
    ↪ adipiscing elit. Nam id pharetra neque. Donec laoreet ipsum tincidunt
```

File5.txt, size 256B:

```
Aenean ultricies urna ut libero semper euismod. Vestibulum id auctor quam. Mauris
    ↪ dignissim sodales neque quis rutrum. Interdum et malesuada fames ac ante
    ↪ ipsum primis in faucibus. Aenean id nisi pretium urna placerat sagittis sed
    ↪  semper turpis. Nulla et.
```

A.2 Test programs

Code listing A.1: The Hilbert module. When DrawHilbert is called, a Hilbert curve is drawn using mutual recursion through procedures **HA, HB, HC, HD**. Note the forward declaration done using **A, B, C, D**.

```
1  MODULE Hilbert;  (*NW 8.1.2013  for RISC*)
2    IMPORT Display, Viewers, Texts, Oberon, MenuViewers, TextFrames;
3
4    CONST Menu = "System.Close␣␣System.Copy␣␣System.Grow";
5
6    VAR x, y, d: INTEGER;
7      A, B, C, D: PROCEDURE (i: INTEGER);
8
9    PROCEDURE E;
10   BEGIN Display.ReplConst(Display.white, x, y, d, 1, Display.paint); INC(x, d)
11   END E;
12
13   PROCEDURE N;
14   BEGIN Display.ReplConst(Display.white, x, y, 1, d, Display.paint); INC(y, d)
15   END N;
16
17   PROCEDURE W;
18   BEGIN DEC(x, d); Display.ReplConst(Display.white, x, y, d, 1, Display.paint)
19   END W;
20
21   PROCEDURE S;
22   BEGIN DEC(y, d); Display.ReplConst(Display.white, x, y, 1, d, Display.paint)
23   END S;
24
25   PROCEDURE HA(i: INTEGER);
26   BEGIN
27     IF i > 0 THEN D(i-1); W; A(i-1); S; A(i-1); E; B(i-1) END
28   END HA;
29
30   PROCEDURE HB(i: INTEGER);
31   BEGIN
32     IF i > 0 THEN C(i-1); N; B(i-1); E; B(i-1); S; A(i-1) END
33   END HB;
34
35   PROCEDURE HC(i: INTEGER);
36   BEGIN
37     IF i > 0 THEN B(i-1); E; C(i-1); N; C(i-1); W; D(i-1) END
38   END HC;
39
40   PROCEDURE HD(i: INTEGER);
41   BEGIN
42     IF i > 0 THEN A(i-1); S; D(i-1); W; D(i-1); N; C(i-1) END
43   END HD;
44
45   PROCEDURE DrawHilbert(F: Display.Frame);
46     VAR k, n, w, x0, y0: INTEGER;
47   BEGIN k := 0; d := 8;
48     IF F.W < F.H THEN w := F.W ELSE w := F.H END ;
49     WHILE d*2 < w DO d := d*2; INC(k) END ;
50     Display.ReplConst(Display.black, F.X, F.Y, F.W, F.H, Display.replace);
51     x0 := F.W DIV 2; y0 := F.H DIV 2; n := 0;
52     WHILE n < k DO
53       d := d DIV 2; INC(x0, d DIV 2); INC(y0, d DIV 2);
```

```
      x := F.X + x0; y := F.Y + y0; INC(n); HA(n)
    END
  END DrawHilbert;

  PROCEDURE Handler(F: Display.Frame; VAR M: Display.FrameMsg);
    VAR F0: Display.Frame;
  BEGIN
    IF M IS Oberon.InputMsg THEN
      IF M(Oberon.InputMsg).id = Oberon.track THEN
        Oberon.DrawMouseArrow(M(Oberon.InputMsg).X, M(Oberon.InputMsg).Y)
      END
    ELSIF M IS MenuViewers.ModifyMsg THEN
      F.Y := M(MenuViewers.ModifyMsg).Y; F.H := M(MenuViewers.ModifyMsg).H;
      DrawHilbert(F)
    ELSIF M IS Oberon.ControlMsg THEN
      IF M(Oberon.ControlMsg).id = Oberon.neutralize THEN
        Oberon.RemoveMarks(F.X, F.Y, F.W, F.H)
      END
    ELSIF M IS Oberon.CopyMsg THEN
      NEW(F0); F0^ := F^; M(Oberon.CopyMsg).F := F0
    END
  END Handler;

  PROCEDURE New(): Display.Frame;
    VAR F: Display.Frame;
  BEGIN NEW(F); F.handle := Handler; RETURN F
  END New;

  PROCEDURE Draw*;
    VAR V: Viewers.Viewer; X, Y: INTEGER;
  BEGIN Oberon.AllocateUserViewer(Oberon.Par.vwr.X, X, Y);
    V := MenuViewers.New(TextFrames.NewMenu("Hilbert", Menu), New(),
      TextFrames.menuH, X, Y)
  END Draw;

BEGIN A := HA; B := HB; C := HC; D := HD
END Hilbert.
```

Code listing A.2: The PrimeNumbers module. When Generate is called, a text scanner is opened reading an input telling it how many prime numbers to generate. Then, the procedure Primes is called, generating the given number of prime numbers and appending them to a text piece. Finally, the primes are output to the Oberon Log.

```
MODULE PrimeNumbers;  (*NW 6.9.07; Tabulate prime numbers; *)
                      (* for Oberon-07  NW 25.1.2013*)
  IMPORT Texts, Oberon;

  VAR n: INTEGER;
    W: Texts.Writer;
    p: ARRAY 400 OF INTEGER;
    v: ARRAY 20 OF INTEGER;

  PROCEDURE Primes(n: INTEGER);
    VAR i, k, m, x, inc, lim, sqr: INTEGER; prim: BOOLEAN;
  BEGIN x := 1; inc := 4; lim := 1; sqr := 4; m := 0;
    FOR i := 3 TO n DO
      REPEAT x := x + inc; inc := 6 - inc;
        IF sqr <= x THEN  (*sqr = p[lim]^2*)
```

```
          v[lim] := sqr; INC(lim); sqr := p[lim]*p[lim]
        END ;
        k := 2; prim := TRUE;
        WHILE prim & (k < lim) DO
          INC(k);;
          IF v[k] < x THEN v[k] := v[k] + p[k] END ;
          prim := x # v[k]
        END
      UNTIL prim;
      p[i] := x; Texts.WriteInt(W, x, 5);
      IF m = 10 THEN Texts.WriteLn(W); m := 0 ELSE INC(m) END
    END ;
    IF m > 0 THEN Texts.WriteLn(W) END
  END Primes;

  PROCEDURE Generate*;
    VAR S: Texts.Scanner;
  BEGIN Texts.OpenScanner(S, Oberon.Par.text, Oberon.Par.pos); Texts.Scan(S);
    IF S.i < 400 THEN
      Primes(S.i); Texts.Append(Oberon.Log, W.buf)
    END
  END Generate;

BEGIN Texts.OpenWriter(W);
END PrimeNumbers.
```

Code listing A.3: Test harness for comparing performance on reading from a file to memory and reading from a sector to memory.

```
MODULE TestFileTransfer;
  IMPORT SYSTEM, Kernel, FileDir, Files;

  TYPE
    HeapLoc = RECORD
      arr: ARRAY 5000 OF INTEGER;
    END;

  VAR filesize: INTEGER;
    h: POINTER TO HeapLoc;

  (* write a file to predetermined sectors to setup sector writing test *)
  PROCEDURE WriteFileToSec(name: ARRAY OF CHAR);
    VAR F: Files.File; R: Files.Rider; heap: POINTER TO HeapLoc;
      adr, word, i, size, sec: INTEGER;
  BEGIN
    F := Files.Old(name);
    Files.Set(R, F, 0);
    NEW(heap); ASSERT(heap # NIL);adr := SYSTEM.ADR(heap.arr);
    i := 0;
    REPEAT
      Files.ReadInt(R, word); SYSTEM.PUT(adr + i, word); INC(i, 4);
    UNTIL R.eof;
    IF word = 0 THEN DEC(i,4); END;
    filesize := i; size := i;
    sec := Kernel.FSoffset + Kernel.mapsize + 1;
    FOR i := 0 TO size BY FileDir.SectorSize DO
      Kernel.PutMemToSector(sec*29, adr + i, size - i - 4);
      INC(sec);
    END;
```

```
31    END WriteFileToSec;
32
33    (* write from file to memory *)
34    PROCEDURE TransferFromFile(name: ARRAY OF CHAR);
35      VAR F: Files.File; R: Files.Rider; heap: POINTER TO HeapLoc;
36        word, i, adr: INTEGER;
37    BEGIN
38      NEW(heap); ASSERT(heap # NIL); adr := SYSTEM.ADR(heap.arr);
39
40      SYSTEM.EBREAK();
41      F := Files.Old(name);
42      Files.Set(R, F, 0); i := 0;
43      REPEAT
44        Files.ReadInt(R, word); SYSTEM.PUT(adr + i, word); INC(i, 4);
45      UNTIL R.eof;
46      SYSTEM.EBREAK();
47      IF word = 0 THEN DEC(i,4); END;
48      h := heap; filesize := i;
49    END TransferFromFile;
50
51    (* write from sector to memory *)
52    PROCEDURE TransferFromSec(sec, size: INTEGER);
53      VAR heap: POINTER TO HeapLoc;
54        word, i, adr: INTEGER;
55    BEGIN
56      NEW(heap); ASSERT(heap # NIL); adr := SYSTEM.ADR(heap.arr);
57
58      SYSTEM.EBREAK();
59      FOR i := 0 TO size BY FileDir.SectorSize DO
60        Kernel.GetSectorToMem((sec + i DIV FileDir.SectorSize) * 29,
61          adr + i, size - i);
62      END;
63      SYSTEM.EBREAK();
64    END TransferFromSec;
65
66  BEGIN
67    (* test writing file to memory *)
68    TransferFromFile("File5.txt");
69    (* test writing sector to memory*)
70    WriteFileToSec("File5.txt");
71    TransferFromSec(Kernel.FSoffset + Kernel.mapsize + 1, filesize-4);
72  END TestFileTransfer.
```

Appendix B

Project Report: Porting the Oberon system to the RISC-V instruction set architecture

Before this thesis was started, a project was carried out to port the Oberon system to RISC-V. While a summary was given in section 2.2.4, the project report attached below explains the process of porting the system in more detail, including a detailed account of some of the issues encountered while porting the compiler, as well as how they were solved. In addition, the results of that project have implications for the results of this thesis, particularly in terms of performance and binary size. In summary, it is of interest in understanding the Oberon system on RISC-V more fully: in the ways it differs and in the ways it does not.

Note that page numbers etc. from the original report are preserved.

NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4501 - Computer Science, Specialization Project

Porting the Oberon system to the RISC-V instruction set architecture

Author:
Rikke Solbjørg

December, 2020

Contents

List of Figures

List of Tables

1 Introduction

A common adage within computer science is that "software is getting slower more rapidly than hardware becomes faster"[5]. As software grows in size and complexity, they become more demanding of hardware, and of their users if they wish to comprehend it. Operating systems are not excepted from this issue, and commonly used operating systems such as GNU/Linux and Windows have seen similar increases in complexity over time.

The Oberon system is an operating system developed by Niklaus Wirth and Jürg Gutknecht, designed to show that this trend is unnecessary. While it lacks many of the niceties of modern operating systems, it is fully featured, including both a file editor and a compiler. It does not contain, say, a web browser, but does have facilities for exchanging files between workstations over a network connection. On the whole, it is very small, and the entire operating system, along with the compiler, occupies less than $200KB$ when compiled[8]. However, it runs on an instruction set architecture (ISA) defined specifically for this operating system; as such, its latest edition is not supported by commercial hardware, instead requiring its users to put a compatible processor on a field-programmable gate array to run it.

This project focused on porting the Oberon system from its own instruction set architecture to another. For the focus of this report, the instruction set architecture mainly defines how the programmer interacts with hardware: the number of registers available, the operations the programmer can instruct the processor to perform, and so on. These decisions have implications for hardware design, but that is outside the scope of this report. Rather, the focus will be on the implications it has for *software*. Here, the targeted instruction set is RISC-V, which has seen increased adoption in recent years as a free alternative to other ISAs.

This is the foundation for the research question this report will focus on answering: *What will have to be done differently to make the operating system function on an instruction set architecture besides its own? Will it run as efficiently, and take up as little space?*

This report will first relate important background and prerequisites to the project in section 2; more detail on the two instruction set architectures discussed in the report is given in sections 2.1 and 2.3.1. Next, the report will answer the first question, i.e. what must be done differently. Section 3 goes into detail on how to emulate both instruction set architectures. The emulator allows for running the ported operating system; the process of porting the compiler as well as the operating system is described in sections 4 and 5 respectively. How to build the ported operating system into an image that can be run by an emulator or by hardware is given in section 6. Finally, to answer the second question, an evaluation of the system in RISC-V compared to in RISC-5 is given in section 7, before concluding in section 8.

2 Background

2.1 RISC-V

RISC-V is a rather new instruction set architecture, the goal of which is to become completely universal[2]. In other words, it should be able to accomodate all possible cores that desire to implement it, whether they are in-order or out-of-order; as well as all technologies a core can be fabricated with, whether it's on an FPGA or ASIC.

It is unique for many reasons, and not all of them will be recounted here. Of particular importance to this project, RISC-V is an example of a RISC (reduced instruction-set computing) design, meaning it favours combining multiple simple instructions to do something complex, as opposed to performing the complex instruction in hardware. This is as opposed to a CISC (complex instruction-set computing) design, wherein the processor understands many more instructions that can perform very specific operations.

Another important aspect to take heed of in this project is that it is a *modular* ISA[2]. It offers a basic set of instructions that every RISC-V processor is guaranteed to implement, and then a set of extensions that a processor can choose to implement depending on what it targets. For instance, a small implementation might use `RV32I`; the RV signifies that it's RISC-V, *32* signifies a 32-bit processor, and *I* signifies the most basic extension, which includes only instructions essentially deemed necessary, 47 in total. Other extensions can be added on top of this; for instance, instructions for multiplication and division are defined in the *M* extension; an implementation that also includes these instructions would be a `RV32IM` processor. There are many other extensions, such as F and D for single- and double-precision floating point respectively.

For this project, a `RV32IM` architecture was targeted, as it is a reasonable architecture for embedded applications, including the kinds of constraints under which Oberon performs well. As mentioned, that means basic instructions, as well as multiplication and division. Notably, it also means that floating-point operations are not supported by hardware.

Another feature offered by RISC-V is various modes that signify different levels of privilege. It offers three: machine-mode, supervisor-mode, and user-mode[4]. While many modern operating systems require different privilege levels, this is not necessary for the operating system targeted in this project. As such, a system using only machine-mode was targeted, often ideal for simple embedded systems[4].

2.2 Lean software

The development of the Oberon system is not driven by the same interests as e.g. GNU/Linux or MS Windows. Niklaus Wirth is, among other accomplishments, famous for his dedication to lean software, as opposed to "fat software". This dedication has influenced much of the decision making in the design process of the

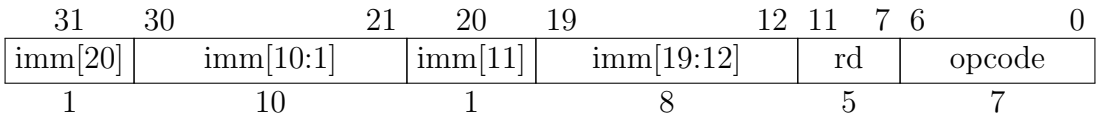


Figure 1: The J-type instruction format in RISC-V[2].

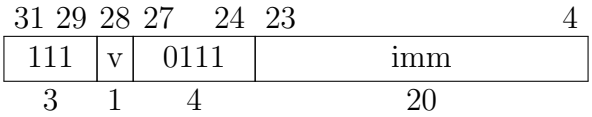


Figure 2: The branch instruction format in RISC-5, with condition set to be an unconditional jump[7]. The v bit controls whether the return address is deposited or not, and can essentially be ignored for the purposes of this figure.

language, compiler, and operating system as a whole, which stands as an example of Wirth's tenets[5]. Among some of the core principles outlined, perhaps of most interest to Oberon is the following: "A system that is not understood in its entirety, or at least to a significant degree of detail by a single individual, should probably not be built." In contrast to modern operating systems, such as GNU/Linux, the Oberon system is small enough that, with enough effort, an individual can have a complete understanding of the entire codebase.

2.3 The building blocks of the Oberon system

Here, the pieces underpinning the Oberon system are explained in some detail. Take special care to note that both the instruction set architecture, the programming language, and the operating system are co-designed, leading to a high degree of compatibility between them.

2.3.1 RISC-5

From 2013 and onwards, Wirth started working on a revised edition of the Oberon system, named *Project Oberon 2013* or simply *Project Oberon*. In this document, it will be referred to as *the Oberon system* to distinguish it from the programming language of the same name. While overall quite similar to previous versions, a major difference in Project Oberon 2013 is the use of a new processor, along with a new instruction set architecture. Rather than use an already extant instruction set architecture – say, the ARM architecture – Wirth opted to design his own[8]. Rather confusingly for this report, the latest and most stable version of this architecture is named *RISC-5*, and the reader must take care not to confuse this with *RISC-V*.

Wirth defined RISC-5 to be as simple as possible, although some care must be made to specify that this simplicity is most easily appreciated from a software perspective, rather than from a hardware perspective. A short, illustrative example will be helpful.

RISC-V was designed to reduce hardware complexity - for instance, the sign bit of

an immediate is always placed in bit 31 in every instruction encoding, to "speed sign-extension circuitry"[3]. This is part of the reason why the J-type instruction in RISC-V looks so much more complex, cf. figure 1. In contrast, the equivalent instruction in RISC-5 looks a lot simpler, cf. figure 2. Indeed, the contiguous immediate is much simpler to work with for e.g. a compiler, but this comes at the cost of disallowing some hardware optimisations. Some of the costs of this increased complexity in immediate encoding in RISC-V are discussed in more detail in section 5.3.

2.3.2 The Oberon programming language

The Oberon programming language, designed by Niklaus Wirth, is currently the latest language in Wirth's family of Algol-like languages[6]. In contrast to e.g. C, it is a far more strictly typed language, with static typing, while still being suited for systems programming. An example of this strictness is that, while the programming language does have pointers, it only allows pointers to records (which are similar to structs in C). Of particular note here, the Oberon programming language is quite minimal, but has enough features that the entirety of the Oberon system can be written in it.

2.3.3 The Oberon system

The Oberon system began development in 1986, with a primary goal to be both (1) a fully comprehensive system, and (2) to be able to be understood as a whole by a single person[8]. In other words, it must be simple enough that all parts of it can be comprehended by a single person all at once, but not so simple that it isn't useful.

The Oberon system accomplishes this by avoiding large amounts of optimisation where it isn't necessary, such that no part becomes so complex as to require a lot of study, and keeping the set of features minimal.

Another important aspect is the fact that in its latest revision, i.e. Project Oberon 2013, the system has been designed alongside its instruction set architecture, RISC-5. This allows it to take some shortcuts both in the design of both the compiler and the operating system, wherein parts of the ISA have been designed for easier use in the Oberon system. This will become more evident in section 4.

2.4 Compiler Construction

In deciding upon solutions to problems that arose in porting the compiler to RISC-V, some effort was taken not to break entirely with the design philosophy behind it. A core component of this design philosophy is an emphasis on simplicity; one can see this both in the programming language itself, as well as in some of the techniques used – and more importantly, some of the techniques that were explicitly decided against.

The compiler performs very few optimisations as a result of this simplicity, as Wirth focused more on the compiler running quickly rather than creating optimised programs. A simple metric, in part popularised by Wirth, is the *speed of self-compilation*[1]. This metric suggests that the only increase in complexity desirable in a compiler is one that improves compilation speed enough that it makes up for the increased code size required to express the optimisation.

3 Emulation

This section will focus on the tools used to emulate the computer running the Oberon system. Although many different emulators could be used, for ease of development a simple one tailor-made for the Oberon system was put in use, created by Peter de Wachter[1]. The basics of how this emulator works are described in section 3.1. Then, the process of making it emulate RISC-V is described in section 3.2.

3.1 RISC-5 emulator

The RISC-5 emulator is fairly simple, and focused on making Oberon run efficiently on another platform, rather than accuracy. For instance, it does not emulate stalling while waiting for input, as that would be very noticeable, but instead skips emulation if it notices the Oberon system continually reading I/O ports without receiving any new input. Similarly, it also does not simulate e.g. memory latency, bandwidth, etc., and instead immediately returns data read from memory without delays.

3.2 Converting the emulator to RISC-V

Making the Oberon emulator work for RISC-V was fairly simple. The only issues that needed solving were to add an emulator for RISC-V, and to make that emulator interface with the rest, so that display, input, etc. was handled properly. At first, a custom emulator was written, mostly to become more comfortable with the RISC-V instruction set. While this worked fine for most cases, it failed in certain edge cases. Therefore, for most of the project, an already existing RISC-V emulator was integrated instead; this to avoid having to debug a compiler, an operating system, and an emulator all at once. For this purpose, a RV32I emulator by Ted Fried[2] was chosen, as its small size made it easy to integrate. This did not take too long, and worked for the most part. The instructions found in the `M` extension for RISC-V also had to be added, but as this extension only adds eight instructions, this did not present a large obstacle. Another issue, that was only noticed when it caused errors in the Oberon system, is that this emulator did not handle LB (load byte) instructions correctly: a LB instruction only loaded the first byte of whichever word

[1]This emulator can be found here: https://github.com/pdewacht/oberon-risc-emu/

[2]This RISC-V emulator can be found here: https://github.com/MicroCoreLabs/Projects/blob/master/RISCV_C_Version/C_Version/riscv.c

it addressed, rather than the actually addressed byte. Although fixing this is trivial, it did cause some unfortunate side effects during the port before it was discovered, where strings were not loaded properly.

4 Compiler

Porting the compiler was a large part of making Oberon work on RISC-V. In this section, I will go into detail on some of the changes necessary to make it work on RISC-V, and in particular, some of the challenges that arose because of the differences in design of RISC-5 and RISC-V.

4.1 Goal for the compiler

The ultimate goal for the compiler was to be able to compile the Oberon System for RISC-V. As such, the main consideration for any feature was whether or not the Oberon System needed it. While the Oberon System does need most features offered by the Oberon programming language, it does not make any use of neither real numbers (i.e. floating-point) nor interrupts. Even though some programs may want or need these features, none of them are necessary for a complete Oberon system. As such, they are not implemented.

4.2 Open-source

Around three weeks into the project, I discovered Samuel Falvo II's version of an Oberon compiler for RISC-V, also based on Wirth's RISC-5 compiler[3]. As this was open-source, and seemed to be a good foundation for porting the rest of the Oberon system, I moved over to using this compiler instead.

However, I discovered that large parts of the compiler did not function quite right, such that many of these edge cases had to be fixed for the full Oberon System to be able to be compiled. This is not too surprising, as a compiler and an operating system are some of the most complex programs one may want to compile. However, this ended up taking quite a lot more time than expected, as bugs in a compiler can be very difficult to track down. Seeing as the compiler is a major part of the process of porting the Oberon System in particular, I will focus on some of the issues faced in making this compiler fully functional.

4.3 Large conditional jumps

A problem particularly influenced by the difference in ISA occurs in large conditional jumps. An example would be an if condition whose body, the code executed if the

[3]This version can be found here: https://github.com/sam-falvo/project-norebo/tree/master/OberonRV

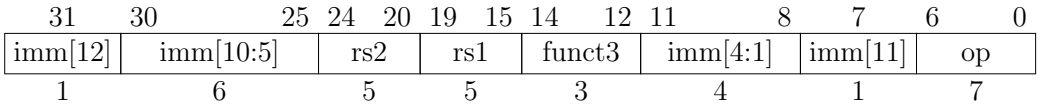
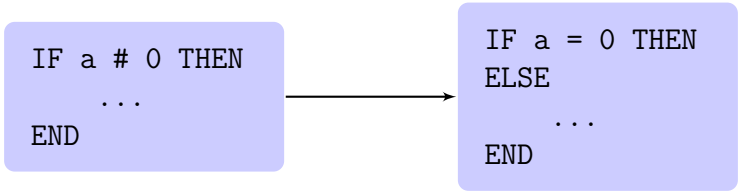


Figure 3: The B-type instruction format in RISC-V.

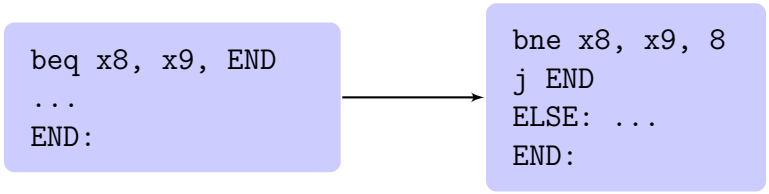
condition is true, spans very many instructions. An implementation of conditional jumps for a single-pass compiler, and the one used by the Oberon compiler, works as follows: place a link for a forward branch; compile the body of the conditional; finally, fixup the previously placed forward branch such that it branches to the end of the conditional's body. (For a more detailed explanation of fixups, see section 5.3, which discusses fixups necessary to link and load modules; this is in principle the same procedure, done during compilation.)

A problem arises when the number of instructions one needs to jump over exceeds the amount of bits allotted to memory displacement in a branch instruction. However, the branch instruction in Wirth's RISC-5 architecture allots 20 bits to memory displacement following a branch, meaning the body of the conditional would have to exceed 524288 instructions before this became an issue - certainly not something that occurs in the Oberon operating system. The roughly equivalent instructions in RISC-V are the branch instructions, with the B-type instruction format, shown in figure 3. Of particular note, then, is the fact that the immediate of the B-type instruction format only allots 12 bits to branch displacement - meaning, due to two's complement, the largest conditional forward jump possible is 2048 instructions, a much lower magnitude. When *Modules* exceeded this number, with an if condition whose body spanned 7512 instructions, this led to behaviour that was difficult to pin down. The resulting instruction's immediate was computed as 7512 mod 4096, which results in a two's complement encoding of -680. In sum, this meant the instruction `bne x8, x9, 7512` was instead encoded as `bne x8, x9, -680`.

The most basic solution for this that was implemented was to, upon fixup, check whether the branch was too large to be possible to branch over, and fire an assertion if this was the case. While certainly not an ideal solution, this allowed the programmer to rewrite offending programs. This rewrite was also quite simple, as one only needed to change a conditional as follows:

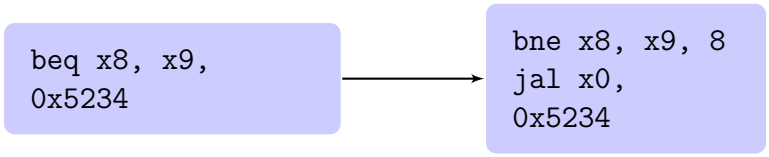


The end of the body of an IF condition will always contain an unconditional jump over the instructions encompassed by the ELSE condition; importantly, unconditional jumps in RISC-V have more bits for offset, allowing larger jumps. Thus, this produces the following instructions:



However, this is hardly a good solution. For one thing, it makes it much more cumbersome to read the program, especially if the programmer also has an ELSE case to take into account. Furthermore, it's something any programmer will assume the compiler ought to handle properly.

A large problem with implementing a better solution, however, was the single-pass nature of the compiler. In a multi-pass compiler, one could for instance replace the offending branch instruction at the head of the if condition with a branch and a jump that can reach farther, like so:



However, with a single-pass compiler, this becomes inordinately clunkier. As it requires two instructions rather than one, it becomes necessary to shift all the code already generated forward to make room for the jump instruction. Furthermore, the list of instructions pending fixups placed within the block that are dependent on the linker – e.g. fixups for retrieving variables from imported modules – must also be rewritten to account for the shift. This was deemed a poor solution, as it is optimising much too heavily for what is a fairly rare case. Alternatively, one could always generate two instructions for fixup, and generate one into a *nop* if it isn't necessary. However, this has a negative impact on performance and code size to, again, optimise for a rare case.

The solution that was eventually put in use is slightly less efficient, but more in-line with what can be easily done in a single-pass compiler. As this is a rare case, using slightly more instructions than necessary in return for much lower complexity in the compiler is a worthwhile trade-off, given some of the principles set out in Section 2.4. Essentially, a few more instructions are appended to the end of the body of the already generated if instruction, that perform the actual branching part; this transformation can be seen in figure 4. This way, the jumps over large portions of code are always performed by jump instructions, which allow for larger offsets.

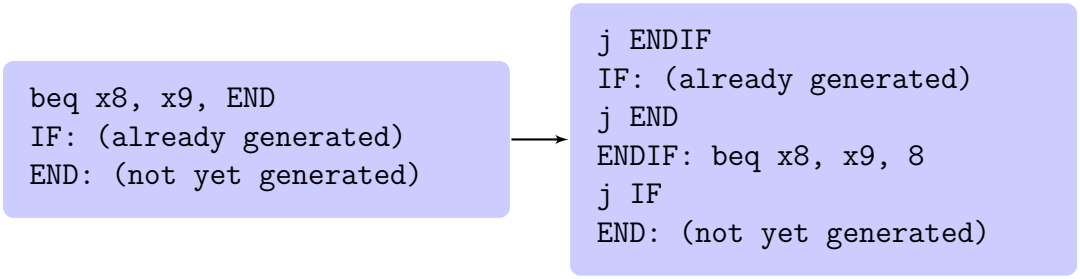


Figure 4: Transformation performed if the instructions generated between IF and END are too large for `beq` to branch over.

4.4 Comparisons

Another stark difference between the RISC-5 and the RISC-V ISAs are in their use of flags. RISC-5 uses *condition flags*, similar to e.g. ARM, wherein a side effect of an arithmetic operation is that it deposits condition codes based on the result in an auxiliary register. For instance, in RISC-5, `sub r0, r1, 2` will deposit a high bit in the auxiliary register N if the result of the subtraction is negative, and similarly, a high bit in the auxiliary register Z if the result is zero. This allows for e.g. determining equality of the two registers. By contrast, RISC-V does not use such condition codes, as by adding extra state they "needlessly complicate the dependence calculation for out-of-order execution"[2] – that is, an instruction can be implicitly dependent on a prior instruction simply because of the condition codes it set. Instead, RISC-V uses branch instructions where the comparison between two registers and the branch from the result of the comparison is done in a single instruction. A quick example of the difference in how the ISAs prescribe branching (note that `cmp` is merely an alias for `sub`):

RISC-5

```
cmp r0, r1
beq 0x5234
```

RISC-V

```
beq x8, x9, 0x5234
```

This has implications for compiler design. In the original RISC-5 compiler, whenever it encountered a comparison, it could immediately perform the comparison, and expect the resulting condition codes to be used by e.g. an upcoming branch. As *performing the comparison* in RISC-5 is simply subtracting the two numbers to be compared, this meant that all relations could be treated similarly, without having to take the actual comparison in question into regard. Regardless of if the comparison is checking for equality or less-than, the subtraction will be the same. This design is well-suited for a simple, single-pass compiler, as a comparison introduces no additional state from the perspective of the compiler. However, as should be clear from the above discussion, this scheme will not work in RISC-V.

The solution for RISC-V, as was already developed by Samuel Falvo II and Peter de Wachter and in use in the open-source Oberon compiler I continued work on (cf.

section 4.2), was to introduce additional state to track which registers to compare in a branch instruction. With the two separate stages of conditional branching – performing the comparison; branching – combined, the state in each separate stage must be known altogether to generate the correct instruction. Two new fields are added to the *Item* object, which carries the state of syntactic objects[8], to track the registers a conditional is comparing; the object was already holding information of what comparison to use.

However, the increased state to track introduces additional complexity, and so is more error-prone. For instance, intermediary calculations affect the register stack, used to allocate available registers; if one is not careful, these can affect the values stored in registers necessary for a later branch instruction. This can cause bugs that are both subtle and potentially system-breaking! A quick example to illustrate this follows.

Oberon supplies a SET data type, representing a 32-bit word upon which operators effectively become bitwise operators; e.g. plus represents OR, multiplication represents AND, etc. Oberon also allows testing for membership of a number in a SET, e.g. `3 IN SectorMap`, which will be true if the fourth bit in the SET is high. Less subtly than checking the relation between two integers, this is still a comparison; in this case, performed by moving the bit of interest (e.g. 3) into the sign bit of the word, and then checking whether this is less than zero. If the value at the location of interest is 1, the word will be negative; if not, it will be positive.

However, an issue that arose from this happened in the case where membership of *the result of an expression* in a SET was tested. Membership of constants and variables worked, but as the result of the expression was located elsewhere on the register stack than the comparison expected, the branch ended up comparing the wrong registers. With luck, this bug would merely be unfortunate; but as the Oberon System used exactly such a comparison to allocate sectors in storage, the file system ended up becoming corrupted when attempting to store a buffer on disk, as it failed to allocate an empty sector, instead allocating an already filled one.

The solution, as opposed to the bug, was fairly simple, and merely required explicitly telling the compiler which register to compare with, rather than relying on its position in the register stack. However, tracking down the bug itself was far more difficult, as such an effect from a failed comparison is not immediately obvious.

4.5 Data section and global variables

This issue is similar to the one discussed in section 4.3, but with a less ideal solution space.

Each module has a separate segment for storing global variables, similar to data segments in ELF binaries. To resolve access of variables in modules' data segments, the linker, discussed in more detail in section 5.3, is needed to resolve such accesses. For access of global variables within the module wherein they're accessed, this is quite simple to resolve: all the linker has to do is fixup an instruction loading the

base of the module's data section into an already specified register already set aside for this purpose.[4] A crucial aspect is that for global variables *within the compiling module*, the offset into the data section for the desired variable is known in compile-time. Therefore, loading a variable with an offset larger than the space possible to reach with a load instruction[5] simply requires loading the offset into a register first. This requires an additional instruction, but this is only generated when necessary.

However, for accessing global variables in other modules' data segments, this solution is less feasible, and a similar decision to the one in section 4.3 has to be made. The issue is simply that the offset to load a global variable from a different module is not known at compile-time, so the compiler cannot generate the instructions required if the offset is larger than the immediate space in a load instruction.[6] However, as mentioned, the solution space here is much smaller: more code *cannot* be generated for the program without increasing the complexity by an order of magnitude, as this would have to be done by the linker, not the compiler. This means, to fully solve the problem, the compiler would have to generate two instructions for every access to an external variable, even if this isn't strictly necessary. The linker could then fixup both of these instructions as required. However, there are several things to consider with this solution:

- It is quite inefficient for most programs, as few modules' global variable space exceeds what offsets can be encoded, worsening performance and size for a rare case.
- This increases the linker's complexity even more, as it now upon any reference to an external module has to fixup three instructions, one to load the base of the module and two to load the variable. As discussed in more detail in section 7, the linker is already a fair share more complex in RISC-V, so this was not desirable.
- The original Oberon System faces a similar issue, although less so, as its offset allows for 64kB of global variables, as opposed to 2kB. However, this difference in size is less stark than it might at first seem: many of the programs that run into issues due to the 2kB limitation on RISC-V can easily run into the same in RISC-5. For example, the array holding all generated instructions in the compiler's code generator, *ORG*, is unable to compile programs larger than 64kB due to this limitation.

An alternative solution could be to put information about offsets in symbol tables, which can be read during compile-time. However, the offsets are likely to change if any part of the program changes, even if no new variables or procedures are exported; this would require creating a new symbol table, and hence recompiling all

[4]Here, this purpose is reserved for x3/gp, often given the purpose of being a global pointer in RISC-V convention.

[5]Currently $2kB$, but easily made $\pm 2kB$ if offset from 2kB into the data segment

[6]For RISC-5, this is not a problem: as the instruction set was designed specifically for the Oberon system (as mentioned in section 2.3), they were able to allot as many bits as required for immediates in their own architecture.

dependent modules. This harms the extensibility of the system, and as such is also a poor solution.

These factors led me to decide against a "complete" solution to this problem, as opposed to the one in section 4.3. Instead, the compiler warns the programmer that external access to a module's data segment may lead to undefined behaviour if it exceeds 2kB. If the programmer requires more than can reasonably fit within the global data space, this can easily be allocated from heap space instead. While only a "workaround", this is beneficial for the Oberon System as a whole: modules are more likely to be able to be loaded; checking whether a memory allocation failed is already standard practice[7]; and it more easily allows for large blocks of data without going over the limit of what can be stored in global variables, including the already extant limit of 64kB in the original Oberon system. Furthermore, it can lead to more efficient use of memory: the garbage collector can free memory in the heap, whereas the data segment is fixed to be as large as need be, occupying space until the user unloads the module.

There are some downsides to this alternative, particularly for the compiler, which was used as a case study for this solution. The compiler allocates memory in the heap upon compilation of a new module, and removes those references when it finishes compilation of the module so the garbage collector can free memory. However, the garbage collector only runs every second; this means that, if the compiler is called again immediately, it will have less memory to work with, as the memory it last allocated to use has not been freed. To circumvent this, one could either reuse allocated memory, implying it must remain allocated for as long as the module is loaded, losing some efficiency of use of memory; force the garbage collector to run between compilation units; or put some span of time between each compilation. The last is currently in use, but the other two can easily be used too.

4.6 State of the compiler

The compiler can compile most programs from within the Oberon System, although not all. The MagicSquares program, which attempts to generate a magic square of the dimensions specified, compiles and runs, but generates invalid magic squares.

There are also a few known imperfections left. There was a bug that led to incorrect reading of symbol tables in compile-time from within the Oberon system. This led to failed compilation on any program that makes a procedure call to an external module, if that procedure expects a CHAR array (i.e. a string) as a parameter, as it read the expected length of the parameter incorrectly; where it should have read `0xFFFFFFFF`, the compiler read `0x7F`. This is likely a result of the compiler only reading a byte where it should read a word, as the value written into the symbol table is correct. Although this bug may sound as if it triggers rarely, it affects any program that attempts to print a string, meaning many programs fail to compile.

One can work around this bug either by compiling all desired modules from outside

[7] Although the Oberon system fails to check if it got a pointer into the heap or merely NIL most of the time, which of course can lead to errors – I have remedied this in my port

the Oberon System using the cross-compiler, which compiles the programs correctly, *or* by adding an edge case to the parser, to make it accept parameters of size 0x7F as equivalent to size `0xFFFFFFFF`. No bugs as a result of this workaround have been observed, and this allows programs to compile successfully, although it leaves the underlying error in the compiler unfixed.

4.6.1 Traps

Another improvement that would be desirable is to make traps more optimal. Currently, traps are performed very similarly to in RISC-5, where a trap cause is deposited in unused bits of an instruction; this is an example of the ISA being designed for the operating system, as mentioned in section 2.3.3. However, because such unused bits do not exist in RISC-V, this is circumvented as follows:

```
jalr ra, mt, 0
jal x0, 0x8
[trap number and position]
```

The kernel, which handles traps, must then look in the instruction after where RA points to find out which trap was triggered and where. The `jal` instruction is necessary to avoid executing the trap number as an instruction, which would fail.

This is not an ideal solution, as it is quite cumbersome and takes several instructions to do what RISC-V *can* support in one. However, as mentioned in section 2.1, the decision to only use machine-mode was made. This becomes a problem for a hardware solution, as RISC-V only supports vertical traps, i.e. traps that increase privilege level. To perform a horizontal trap – a trap within the same privilege level – one must perform a vertical trap, and then return control to a trap handler in the lower privilege level[4]. This is impossible without multiple modes, which would complicate many other parts of the operating system. As such, the current solution is kept.

5 Porting Oberon to RISC-V

5.1 Bootloader

Porting the bootloader of Oberon to RISC-V, or indeed to any platform in which a compiler is already implemented, is surprisingly easy. This is due to two factors: the rather simple assumptions the Oberon system makes about hardware, and compiler support for writing bare-metal software.

Firstly, as the Oberon system does not rely on neither interrupts nor an MMU, configuring these in startup code is not necessary. Furthermore, the Oberon system effectively assumes the entire operating system is run in machine-mode, so entering

user mode is at no point necessary either. In other words, no aspect of booting the Oberon system necessitates writing assembly code.

Secondly, the Oberon compiler has specific syntax for compiling to bare-metal code. In other words, the compiler can compile a program written in Oberon, and so long as the program does not rely on any part of the Oberon system or make use of abstractions that necessitate an operating system, such as heap memory allocation, it will compile into machine code that can be run without an operating system. Additionally, the compiler places a jump to memory location 0x0 in the end of execution of bare-metal code, meaning the build of the operating system in storage being loaded must have a jump into its entry point there.

The result of these two factors is that the bootloader for the Oberon system can be written *entirely* in Oberon. No handwritten assembly code is necessary. The bootloader consists in large part of a driver that can read from an SD card, as its only purpose is to load the necessary modules from storage into main memory such that the Oberon system can boot.

To be certain, some modification of the bootloader is still necessary: for instance, it needs to set the value of the stack pointer, and the register of the stack pointer will differ depending on the instruction set architecture. Furthermore, if the system does not boot from an SD card, a new driver will have to be written to support the desired storage system. However, in this project, the assumption is a similar system aside from the change in ISA, and as such the driver itself remained untouched.

5.2 Kernel

The kernel in the Oberon system is responsible for a few things – though not nearly as many as a monolithic kernel would. It serves four purposes: it organizes memory, taking care of memory allocation in the heap; it contains a device driver for the SD card; it has a trap procedure, for use before the outer core of the Oberon system has been loaded; and finally, it contains some miscellaneous procedures for tracking time. Neither the device driver nor functions for returning time need elaboration, as they are unchanged between RISC-5 and RISC-V. However, the trap procedure must be treated somewhat differently, as the Kernel is responsible for placing a jump instruction into the trap procedure where the compiler expects it to be. In essence, the compiler places a jump into a trap vector table, and the kernel is responsible for placing a jump to the trap procedure into that vector table.

5.3 Linker and loader

A more complex part of the porting process was fixing the linker and loader in the Oberon system. These are part of the *Modules* module.

It must be stressed that this is a separate step from porting the compiler. Upon encountering any Oberon code that relies on external references - e.g. calling a function that resides in another module - the compiler, rather than generating working

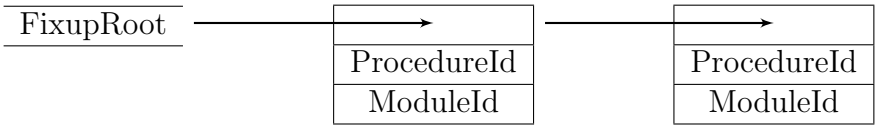


Figure 5: The linked list traversed by the linker to perform necessary fixups. The compiler constructs the list, and the linker changes every instruction within it into a valid RISC-V instruction upon traversal.

RISC-V code, leaves a word indicating what object it is referencing, as well as where the next instruction in need of linking in the module is. This process is known as a *fixup*, and the linked list of instructions formed within the program is a *fixup list*. The header of the program stores a reference to the first element of each fixup list, which the linker can then traverse, changing every instruction found in the list to match what the programmer intended; this mechanism is shown in figure 5. In the process of doing this, external references go from using addresses relative to the base of the module, meant for interpretation by the linker, to absolute addresses that work in the runtime system the program is meant for.

This job can be done by a *static linker*, which would package all the dependencies of the program into the final binary, creating a standalone binary. This is an older technique, although it still has its uses today to create binaries without assumptions about what libraries exist on the target system. The Oberon system instead uses a dynamic linker, i.e. a linker that links program dependencies in runtime. There are several reasons for this choice[8]:

- It avoids recompilation of a module if one of its dependencies has changed.
- Improved extensibility: new modules can more easily depend on modules that have already been loaded.
- Space: statically linked binaries are often much larger than dynamically linked binaries, which becomes a greater concern on a system with 1MB of RAM.

As mentioned, the *Modules* module takes care of this, and is in fact the first module entered from the bootloader. What changes were necessary to make this work in RISC-V, then?

Large parts of the linker and loader could remain untouched. Traversal of the lists, for instance, worked the same both in the old system as well as in the RISC-V port. The change necessary was largely in formatting the fixups such that they would become valid RISC-V instructions. Of particular importance here is the fact that encoding RISC-V instructions in software is more difficult than RISC-5 instructions, as discussed in 2.3.1.

```
SYSTEM.PUT(adr, (offset MOD 1000000H) + 0F7000000H);
```

Rewritten for RISC-V

```
(* Returns a `jal ra, imm` instruction. *)
PROCEDURE Jal(imm: INTEGER) : INTEGER;
  VAR imm20, imm19to12, imm11, imm10to1: INTEGER;
BEGIN
  imm20 := imm DIV 100000H;
  imm19to12 := (imm - imm20 * 100000H) DIV 1000H;
  imm11 := (imm - (imm20 * 100000H + imm19to12 * 1000H)) DIV 800H;
  imm10to1 := (imm - (imm20 * 100000H + imm19to12 * 1000H + imm11 *
    800H)) DIV 2H;
  RETURN ((((imm20 * 400H + imm10to1) * 2H + imm11) * 100H +
    imm19to12) * 20H + 1) * 80H + 111
END Jal;
(* ... fixup code here ... *)
SYSTEM.PUT(adr, Jal(offset MOD 200000H));
```

Figure 6: RISC-V increases complexity of performing fixups. Shown here, an equivalent fixup of calling functions in external modules, in RISC-5 and RISC-V.

6 Building the Oberon system

In this section, I will go into some detail on how to build the Oberon system for RISC-V. Currently, this has only been done targeting an emulator, but there is nothing suggesting it should be very different in hardware, aside from changing device drivers to target different I/O ports.

6.1 Project Norebo

The build system used throughout this project was based on Peter de Wachter's Project Norebo[8], a free and open-source project to run programs intended for the Oberon system on POSIX systems. It does this through emulation, using the RISC-5 emulator discussed in section 3.1, but redirecting text and file output to the POSIX system it runs on rather than in the Oberon system. In addition, it offers a Python script to build images, and the images built by it can also be run by the RISC-5 emulator. The images it generates can also be installed on SD cards, to be used on an FPGA running an appropriate core; this has not been tested yet, but should in theory work. This build system allows for building a complete Oberon system from a GNU/Linux system, which is much faster than building it from an emulated full

[8]The source code for the original Project Norebo can be found here: https://github.com/pdewacht/project-norebo

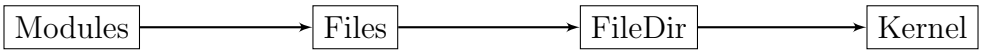


Figure 7: The *inner core* of Oberon, which must be linked beforehand. Arrows show dependencies, such that Modules depends on Files, etc.

Oberon system would be.

To make Norebo build a RISC-V system instead of a RISC-5 system, a few steps were necessary. Most obviously, it had to use the RISC-V compiler, discussed in section 4, as a cross-compiler. This means the compiler was built to target a *RISC-5* system, such that it could run in Norebo's emulation layer, and compile the rest of the files desired to RISC-V, to be run in a RISC-V emulator or system. This change was rather simple, and only required adding another set of functions for RISC-V building, pointing it to the RISC-V compiler rather than use the RISC-5 compiler.

The linker used by Norebo, a module named *CoreLinker*, also required changes to work with RISC-V. To allow the build system to create images for both RISC-V and RISC-5, a module named *RVCoreLinker* (where the RV signifies RISC-V) was created, to link RISC-V programs. One might be tempted to ask why a linker should be necessary here, since the Oberon system dynamically links its modules in runtime. This is on the whole true, but the *Modules* module, responsible for dynamically linking and loading programs, discussed in more detail in 5.3, has a dependency on the *Files* module, to retrieve the files it loads. As it cannot link itself in runtime, this particular module must be linked by the build system. As this is at the top of the chain of dependencies of Oberon's *inner core*[8], of which the full chain is shown in figure 7, this entire part of the Oberon system must be prelinked in the build stage. The bootloader is responsible for placing the linked inner core into memory contiguously, such that the already linked memory references are correct.

Norebo was extended as Oberon was ported, according to the needs demanded by the project. Functionality to generate images that only loaded a particular module to be tested proved to be worthwhile. Similarly, flags to define whether to build a RISC-V or RISC-5 image, as well as a flag to use a different manifest (list of files to build), were useful to quickly build different versions of the operating system. Norebo was also extended to, after installation, verify whether files were correctly installed to the image, which proved itself useful when e.g. accidentally using the wrong version of a manifest.

The end result of making Project Norebo work for RISC-V is that, as mentioned, building RISC-V images becomes faster and easier; all the commands necessary to create a new build are:

```
rm -rf imagebuild; ./build-image.py -r OberonRV/Oberon
```

A disk image along with all compiled programs and symbol tables are deposited in the `imagebuild` folder.

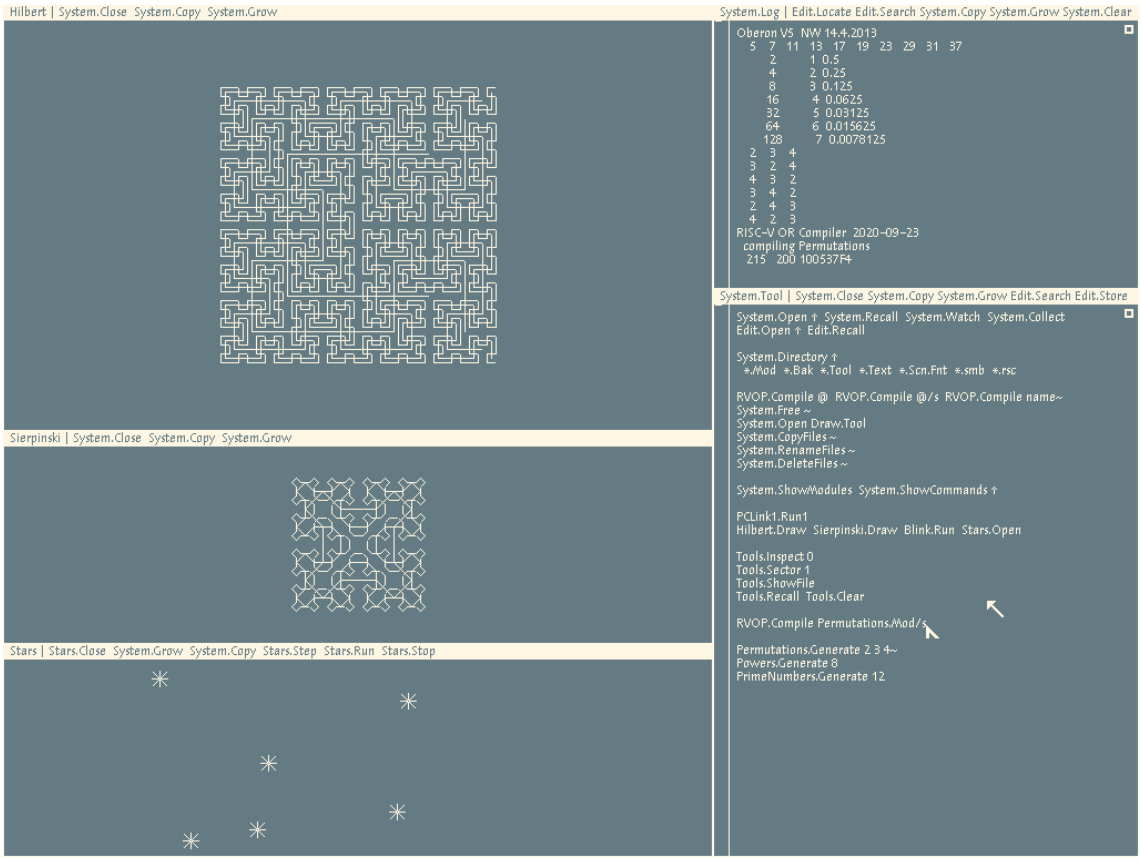


Figure 8: The Oberon system, running in a RISC-V emulator.

7 Results and Discussion

Some interesting results can be gathered from the RISC-V port, when compared to the original. These can give some insight into benefits of RISC-V over the previous ISA used by the Oberon system, RISC-5.

7.1 Working Oberon system

The major result of the project is a version of the Oberon system that runs on RISC-V, and is close to fully functional, excepting bugs discussed in 4.6. This can be seen in figure 8. It can run most programs without issues, and supports the same set of features as the RISC-5 version.

7.2 Quantitative results

As mentioned in section 3.1, the emulator's main focus is speed and ease of use, not accuracy. As such, the data that can be gathered is rather limited, and would require either a different emulator or a hardware platform. However, some data can be gathered with the tools at hand, mostly some notes on speed and size, which will be examined in more detail below.

To gather information on traces of specific parts of programs, "breakpoints" were left at the beginning and end of the area of interest. This is done in the form of writing specific magic numbers to the I/O port that writes all input to LEDs, which can then be identified in logs from the emulator, so that only instructions run between the beginning and end of the area of interest are included.

7.2.1 Instructions run

A metric to consider is the amount of instructions it takes for the system to boot. *Booting* is here considered not to be just the time spent in the bootloader, but as the time spent from turning on the system and entering the bootloader, to entering the main loop in the *Oberon* module. Table 1 shows the difference in instructions performed to do this. As one might expect going from one RISC architecture to another, the difference is not too large, but still notable, with RISC-V taking 8.63% more instructions to fully boot than RISC-5.

Number of instructions run	RISC-5	RISC-V	Increase in RISC-V %
boot	7664855	8326033	8.63%
boot (only linker)	450092	561861	24.83%
boot (only linker and files)	4334435	5146595	18.74%
Hilbert.Draw (with linker and files)	1881046	1831383	-2.64%
Hilbert.Draw (without linker and files)	1610140	1534963	-4.67%
PrimeNumbers.Generate 12 (with linker and files)	84006	92634	10.27%
PrimeNumbers.Generate 12 (without linker and files)	10150	9696	-4.47%

Table 1: Comparison of the amount of instructions it takes to boot the Oberon system on RISC-5 and RISC-V respectively. In addition, some standalone programs are included.

The entire process of booting the Oberon system is shown in figure 9, which shows where the processor is executing instructions over time. The bootloader is excluded from this figure, as it resides in the memory region of $0xFFFFF800 - -0xFFFFFFFF$, and as such would vastly reduce the legibility of the figure. From this graph, one can ascertain which parts are run for the longest period of time during booting: particularly notable is the block that takes up most of the execution time, from time $1.33e6$ to $7.49e6$. This period of time is almost entirely occupied by the linker and loader, as well as calls to the file system. The "spikes" into higher memory regions seen in this period are due to initialization bodies of loaded modules being called, which are noticeably short.

As a very large part of booting is linking and loading the rest of the operating system, the increased complexity of linking seemed like a good reason for the increased time spent in RISC-V. To confirm, a trace taken of *just* the time spent linking was taken.

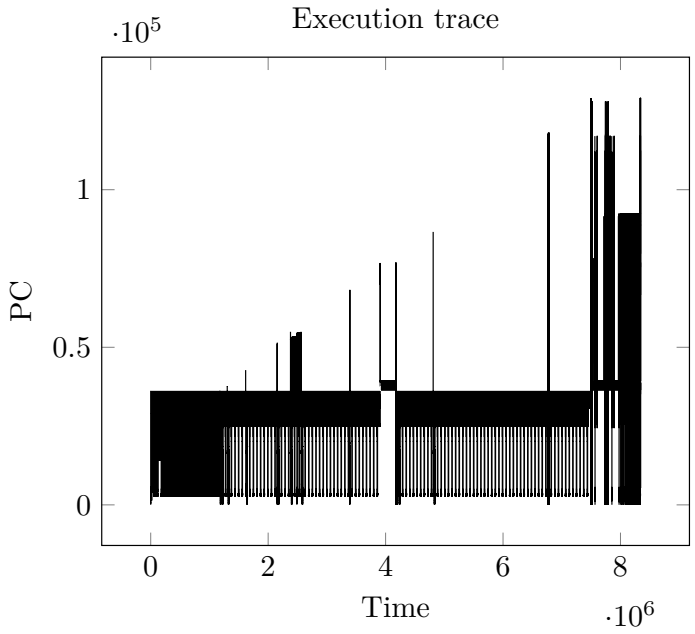


Figure 9: Execution trace of the Oberon system, taken from after the bootloader transferred execution to the operating system to when it finally enters the scheduler loop for the first time. Each timestep is the execution of a single instruction.

As can be seen from the row *boot (only linker)* in table 1, loading and linking the outer core almost takes 25% more instructions to do in RISC-V. The reason for this is likely largely that the linker is more complex for RISC-V, as already discussed in section 5.3. However, while responsible for part of the increased boot time, it is not responsible for all of it. If one includes calls to the file system into this as well, included in row *boot (only linker and files)*, one gets a more complete picture of where the increase in time comes from. Note that this does not include the loading of the files themselves, for which the Kernel is responsible, as that is part of the driver to the SD card. The file system has not seen any significant revision to work in RISC-V, but is rather complex, performing many operations that are less optimal on RISC-V. However, note that the difference in boot time is *less* than the difference in linking, meaning the RISC-V port saves instructions elsewhere.

For more insight, data was gathered on standalone programs as well, namely Hilbert and PrimeNumbers, which respectively draw Hilbert curves and generate prime numbers. These are particularly interesting for this purpose as they can compile without modification for both RISC-5 and RISC-V, due to their lack of low-level system calls, while still interacting with the operating system. This means they are more representative of speed-ups or slow-downs that are not caused by modifications to the program, but rather to the compiler. In addition, to get a fuller picture of the cost of the linker's complexity as well as potentially increased operating system overhead, time spent from when the programs start linking is also included in table 1.

Evidently, both Hilbert and PrimeNumbers[9] spend *less* time executing the core of

[9]Both of these programs can be found in Wirth's Project Oberon repository: https://people.inf.ethz.ch/wirth/ProjectOberon/. Note that PrimeNumbers is found in `SmallPrograms.Mod`.

the program, but *more* time in linking, loading, and operating system overhead.

What, then, causes the programs themselves to spend less instructions running to completion? The compiler does not particularly optimise for using the zero-register (although it quite easily could). This result is somewhat surprising, as it is not immediately obvious what causes the RISC-V program to require running less instructions. Upon closer inspection, the difference is mostly due to the fact that branching in RISC-5 takes two instructions – one to compare, one to branch – while in RISC-V it only takes one, as discussed in closer detail in section 4.4. While other programs may take more instructions in RISC-V, these have many small loops and branches, which favours RISC-V. This also seems to be where RISC-V saves instructions elsewhere in the booting process: many parts of the rest of the booting process, e.g. the Kernel, consist of many such small loops.

7.2.2 Binary size

As the Oberon system should function with limited memory – with the latest revision being created for an FPGA with 1MB memory – keeping modules small is rather important. A comparison of the size of programs compiled for RISC-5 and RISC-V respectively are found in table 2.

Program	RISC-5	RISC-V	Increase in RISC-V %
Hilbert	2397B	2873B	19.86%
PrimeNumbers	995B	1107B	11.26%
Modules	5675B	6587B	16.07%
RVOG	33241B	35753B	7.56%

Table 2: Sizes of different binaries compiled with the RISC-5 and RISC-V compilers respectively. For Modules, the linker for RISC-V was compiled in both cases; i.e., they compiled the same program.

Binaries compiled for RISC-V are generally larger, though not prohibitively so. This is for several reasons: instructions performing on immediates often have to spend multiple instructions rather than one, if the immediate is large enough that it has to be loaded into a register to be operated on; traps require three instructions rather than one, as discussed in section 4.6; etc. Note that despite both Hilbert and PrimeNumbers being larger binaries, they still perform better than in RISC-5 once invoked, as the program does not spend most of its time in code that has expanded to more instructions from RISC-5 to RISC-V.

7.3 Future work

There are still things that can be done for this project, although the main focus was accomplished. For one, the Oberon system has not been tested on hardware, although this should be feasible to accomplish without too much trouble. Additionally, there are still some bugs and inefficiencies left in the compiler, as discussed in

section 4.6, although they are minor enough not to have any impact on the operating system or the compiler. Finally, to fully support all software developed for Oberon, interrupts and REALs need to be implemented, although they are not for the moment. Both should be feasible; REALs especially if one targets hardware that supports floating-point operations. If not, a good solution would be to make the compiler generate system calls to the operating system for floating point emulation.

In addition, the Oberon system has the potential to run on more limited hardware than it is currently targeting. The execution trace of Oberon shows large parts of loaded code being run once, and never again; this can be resolved more optimally by using overlays to transfer these blocks of code out of main memory when they are no longer needed, and load them back in if they are needed once more. This could improve the Oberon system's capability to run on smaller systems.

8 Conclusion

In this project, the Oberon system – meaning the compiler as well as the rest of the operating system – was ported to run on RV32IM hardware, using only machine-mode. As a result of this effort, an emulator that provides an easy way to test the Oberon system was extended to support RISC-V; and a build system, Project Norebo, was also extended to create RISC-V images.

In the process of porting the system, the compiler in particular had to go through some extensive changes to work properly. This was in part due to RISC-V being somewhat more complex to work with in software, due to more complicated instruction encoding; and in part due to some fundamental differences between the two ISAs, as one uses condition codes while the other compares registers upon branch.

Another aspect of RISC-5 that became clear in the process, and made the porting process to RISC-V more complicated, is the fact that it was designed explicitly for the Oberon system. For instance, traps in Oberon only take one instruction to jump to the trap vector table, with the instruction also encoding both the character of the program wherein it trapped, as well as the cause of the trap. This is because the jump instruction in RISC-5 includes several unused bits in the word, specifically to make this so efficient. RISC-V, designed to be as general-purpose as possible, naturally lacks unused bits for this purpose.

However, despite RISC-5 being specifically designed for the Oberon system, the results are still promising. RISC-V does not take much longer to boot, and for some programs, it actually requires less instructions to run. As booting only takes 5% more instructions in RISC-V than RISC-5, the difference would likely be imperceptible.

The port has also put some of Oberon's strengths further into view: for instance, while the compiler required much work to create RISC-V programs rather than RISC-5, the rest of the operating system was quite simple, requiring relatively few changes. Much of this comes down to the simple assumptions the Oberon system makes about its hardware: it requires no memory protection, its traps do not rely

on hardware support, and neither floating-point nor interrupts are required for the system to run.

In conclusion, while RISC-5 does offer some helpful simplicity for the Oberon system, it is far from necessary for the system to run well. The differences between RISC-5 and RISC-V offer some challenges in the effort of porting from one to the other, but none of them are insurmountable. The results in terms of performance in RISC-V are also of comparable order of magnitude.

Bibliography

[1] Michael Franz. 'Oberon - The Overlooked Jewel'. In: *The School of Niklaus Wirth*. 2000, pp. 41–54.

[2] David Patterson and Andrew Waterman. *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, 2017.

[3] Andrew Waterman and Krste Asanovic. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. 2019.

[4] Andrew Waterman and Krste Asanovic. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. 2019.

[5] Niklaus Wirth. 'A Plea for Lean Software'. In: *Computer* 28.2 (1995), pp. 64–68.

[6] Niklaus Wirth. 'Modula-2 and Oberon'. In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. 2007, pp. 1–10.

[7] Niklaus Wirth. *The RISC Architecture*. 2018.

[8] Niklaus Wirth and Jürg Gutknecht. *Project Oberon*. Addison-Wesley Reading, 2013.

Rikke Solbjørg

Evaluating Code Overlays on the Oberon System Using RISC-V

NTNU
Norwegian University of
Science and Technology