Karlsruhe Institute of Technology

Aggressive Constant Propagation For Specialized Unikernels

Master's Thesis
submitted by

Marco Schlumpp

to the KIT Department of Informatics

Reviewer:	Prof. Dr. Frank Bellosa
Second Reviewer:	Prof. Dr. Christian Wressnegger
Advisor:	Dr.-Ing Marc Rittinghaus

1. February 2022 – 1. August 2022

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any source or auxiliary means other than these referenced. This thesis was carried out in accordance with the Rules for Safeguarding Good Scientific Practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, August 1, 2022

Abstract

Unikernels promise to enable building specialized combinations of an application and a kernel. Because they are usually built with a specific use-case, they can remove large parts which would be integral part of a classical operating system. While they only support running a single application, they still have to support a wide variety of potential applications to remain useful. There has been a focus in recent years on improving the specialization aspect by allowing the developer to configure included functionality within the unikernel. However, the applications ported to run as part of a unikernel often still rely on configuration text files parsed during startup at runtime.

Because this configuration is not known to compiler, optimizations such as constant propagation or dead code elimination cannot specialize the application itself for the desired use case. We propose a mechanism to determine the effective configuration at runtime and inject the configuration values at compile-time as constants. This allows the compiler to take the actual values into consideration during optimization.

Contents

Chapter 1

Introduction

Unikernels have gained popularity in recent years, because they allow building highly specialized appliances. Compared to classical operating systems they take a more minimalist approach by only supporting running a single application. This allows them to offer performance, size and security benefits.

Unikraft [12] improves upon other unikernels by splitting the code-base into micro-libraries. The developer can add, remove or replace libraries depending on the use case and achieve a highly specialized unikernel for the desired application.

However, a remaining problem is to specialize the application itself to the actual use case. Most applications are built in a way to cover many possible use cases and allow a detailed configuration using text files. These text files are parsed during the application startup and the resulting values are stored in structure accessible from the rest of the application. For applications running in unikernels these files are often bundled as part of the image. This also means that the whole image is usually rebuilt as soon as the use scenario changes and therefore the configuration is updated. Effectively, the compiler could view the values in the configuration structures as constants. This would allow optimizations such as constant propagation and dead code removal to apply on more code.

The goal of this work is to determine the constants based on the runtime behavior of a unikernel. Then we want to insert the discovered constants into the compilation process to allow the compiler to aggressively optimize the resulting unikernel for the used configuration.

Chapter 2 gives an overview over involved concepts. We continue with an analysis of our objective in Chapter 3. With the gained insights we describe the design of our optimization in Chapter 4. Chapter 5 describes our implementation for Unikraft and Clang. After evaluating our implementation in Chapter 6, we conclude this work with a short summary in Chapter 7.

Chapter 2

Background

We give an overview of unikernels, which are the base of this work, in Section 2.1. For our analysis we have to first gather data about a unikernel's execution. This was done using full system simulation with memory tracing, which we will introduce in Section 2.4. We use debugging information to interpret the raw tracing data. Section 2.3 introduces the general concept of debugging information and presents DWARF, the format we used for our work. To implement our approach we have to modify a compiler and introduce the relevant concepts in Section 2.2.

2.1 Unikernels

Traditional operating systems offer a broad set of functionality such as process isolation and hardware abstraction. This is necessary for multiprocess workloads usually found on desktops or bare-metal servers. Today many server workloads in cloud environments consist of a single application such as a database server running in a virtual machine. These bundles of operating system and a few applications are often reused and redeployed across multiple hosts. For these scenarios the huge breath of functionality provided by modern operating systems such as Linux is less important. The application usually only use a small portion of the provided operating-system libraries and kernel features. The components taking care of abstracting the underlying hardware are also less necessary. The hypervisor already exposes a uniform and simplified interface of common resources such as storage and network. The remaining multiprocess functionality in classical operating systems is necessary because most uncritical and optional functionality is offloaded into system processes. This includes basic features such as establishing network connectivity or handling device hot-plug events.

These dormant features still caused trade-offs in the design of operating systems even if they are not used or are not important anymore. For example, UNIX-like operating systems have the concept of users with associated permissions and this has to be considered in most components such as the virtual file system or networking.

Parts of the resulting operating system binaries can be attributed to these features. Removing them would result in smaller images, which would require less disk space and reduce the boot time [15]. Furthermore, including code and features that are not strictly necessary can increase attack surface. A well known example was a vulnerability in OpenSSL's implementation of the heartbeat extension. The offending piece of code was rarely used in normal practice but was remotely exploitable and had a large impact on the internet [6]. Furthermore, features such as privilege separation and the implied system call mechanism can also impose a performance overhead [12, 13].

In recent years, the function-as-a-service model has gained popularity. The cloud providers provide a managed execution environment for programs instead of normal virtual machines. From the customers point-of-view any deployed programs run on-demand for an incoming request. The benefit for customers is that he has to only pay for the actual usage and the application can automatically scale to the incoming requests. In the background these services are implemented on classical operating systems and virtual machines [27]. To achieve low latency the providers keep created instances around and only the first request that creates the instance suffers from a higher latency. These idle virtual machines still take up memory resources which could be used for active instances.

Library operating systems attempt to address these issues by creating a specialized combination of kernel and application. Instead of providing a platform for applications to run on, they are designed as a library to be linked into the target application resulting in a unikernel. The output is an image that contains both the application and kernel portion in a single address space. This allows some optimizations compared to a classical kernel with an application running on top of it. For example, all code runs in the same privilege level and therefore there is no need to execute costly switches between levels. Instead of having to rely on a system call mechanism to communicate with the kernel, the application can directly call the corresponding functions. Depending on the compiler configuration this can also allow optimization across these function calls, which are normally not possible across the system call boundary. Because there is only a single address space, a unikernel does not have to manage multiple address spaces. While having separate processes with different address spaces is a typical feature of classical operating systems, modern applications rely less on this feature nowadays and prefer threads.

Some unikernels such as MirageOS [14], HalVM [9], runtime.js [23], includeOS [10] are designed for running applications written a specific programming language on top of a hypervisor. They usually provide a custom API and therefore porting existing applications written in the fitting programming language can require a significant effort. Software written in an unsupported programming language cannot run on these unikernels at all.

Some unikernels target more specialized use-cases. For example, ClickOS [16] implements a programmable appliance for network middleware elements. Instead of running Click on a standard memory-heavy Linux system with long boot times, Martins et al. built Click on top of Xen's MiniOS and newlibc. Redoing this for every application is time-consuming and requires expertise across the full stack.

In between classical kernels and highly specialized appliance unikernels, are general-purpose unikernels such as Hermitux [18], Drawbridge [19], Graphene [26], and rumprun [22]. They provide a more-general set of features and focus on running unmodified applications. Hermitux, Drawbridge, Graphene are also ABI compatible to a classical operating system, allowing them to run existing binaries without recompilation. While the executed applications can profit from fast boot times, a small memory footprint and a well-isolated environment, they are still limited by the usually rather high-level APIs of the original operating system. These unikernels are built in a monolithic way that all supported applications can be used with the same core build or are based of existing general-purpose operating systems. The individual components of these unikernels not replaceable or removable. For example, it is not possible to replace parts of the booting process or remove support for threads.

Unikraft [12] attempts to further improve the modularity of unikernels. Most of the functionality is encapsulated into micro-libraries, which users can mix and match depending on their requirements. For example, they might choose a low-level block based storage interface for more fine-grained control and higher throughput, but use a standard POSIX-compatible libc to be able to easily port the application. Some base-functionality and abstraction micro-libraries are included in the core repository, but it is easy to replace and add external libraries into the build process. Unikraft uses the Linux's Kconfig system to allow customization of the unikernel. Each micro-library includes a specification in the Kconfig language about the configurable options and their dependencies to potential other configuration options. The application itself is also represented as a micro-library, that can expose options itself. If specified correctly, these options will automatically enable dependent options.

2.2 Compilers

Compilers take a piece of source code and translate it into a usually more low-level language such as machine code. A simple C compiler might simply take each statement in the source program, decompose it into multiple basic operations and output a sequence of machine code instructions for each of them. Usually doing so would rather inefficient code for various reasons. Firstly, the code that developers write are usually not optimized for performance but rather for readability, and extensibility. For example, some sub-expression in an equation might be duplicated for readability reasons. Secondly, the source code lines in high-level languages do not map directly to machine code instructions. One line of code often has to be represented by multiple machine code instructions. In some cases it is possible that a single machine instruction can describe a complex operation (e.g., the x86 `popcnt` instruction, which counts the number of set bits in the register). While some of these instructions can be provided as intrinsics in system programming languages, the result is often non-portable and less readable code. An important feature of high-level languages is that the developer can keep an arbitrary count of variables around. But there are only a limited amount of hardware registers available and therefore not all variables can be kept in registers at the same time. The previously proposed C compiler would have to write the results back to memory for each operation and would not be able to pass the operands around efficiently via registers.

To solve these issues compilers integrate a large variety of optimizations. Modern compilers are roughly separated into three blocks: front end, middle end, back end. The front end parses the source code and transforms it into an intermediate representation. This representation is then optimized using a set of passes. Finally, the back end generates machine code for the target architecture. The back end phase usually also contains some machine specific optimizations. The exact organization can vary by compiler and often compilers subdivide each of these phases into smaller steps. For example, some compilers there are multiple different intermediate languages involved [8].

Sometimes, there are expressions in the program that always evaluate to the same value. This can happen because all operands are constants or the compiler can prove that the input does not matter. In this case, the compiler can apply *constant folding* and evaluate the operation at compile-time, resulting in another constant. *Constant propagation* is a related optimization that allows replacing the usage of constants in other expressions with the constant. These two optimizations are applied until no further constants can be propagated or expressions folded.

Developers often split program logic across multiple smaller and easier to understand functions. In some cases, they may also decide to abstract the concrete function behind an indirect function call. In this case the callee can vary at runtime and the compiler does not know the target. Furthermore, the compiler would not be able to apply for example constant propagation and folding optimizations across these call boundaries.

In some cases the parameters of a function call are known to be constant at the call site. When the compiler optimizes the function itself it only sees a variable.

Therefore, for instance the constant propagation optimization cannot do anything. For these situations, the compiler can inline the body of functions at the call-site. The compiler uses a heuristic to decide what functions to inline. In practice, these heuristics have to find a compromise between the benefits of inlining and the increased code size [4].

The possible interactions between these optimizations mean that applying one optimization can also have an impact on other optimizations. For example, after taking the call site arguments into account when propagating constants, the compiler might be able to eliminate dead code rendering the function small enough to allow inlining the function.

Object-oriented program heavily rely on inheritance and virtual function calls to realize abstractions. In the general case, there can be multiple concrete callees and only during runtime the actual target is known. There are analysis algorithms such as RTA or XTA [25] that can determine a set of potential callees based on inheritance hierarchies. In case only a single candidate exists, the compiler insert a direct call to the determined function. In the C programming language this paradigm is often replicated using function pointers and indirect function calls. Because the previously mentioned algorithms need an inheritance hierarchy they are not used in C compilers.

The processor has only a limited amount of cache for code and the cache is usually organized in larger blocks. When a block of code is fetched, the block might contain both frequently and rarely executed instructions. Therefore, it is possible to make more efficient use of the cache by reducing the amount of *cold* code next to *hot* code. However, the compiler often does not know how frequent a branch is taken as it usually depends on external input. In these cases, the developer could provide explicit hints to mark a branch as less or more likely. This manual process is tedious and as the software evolves these annotations can be outdated resulting in less efficient code than a non-annotated version. Profile-guided optimizations attempt to automate the annotation process by observing the program at runtime. Classically, this involves instrumenting the considered program with counters how often a branch is taken. The program is then executed with "realistic" input data. But in practice, developers run test suites and unit tests, which are often close enough to real-world scenarios to gain meaningful improvements. In a second step, the program is recompiled with the profile data as an additional input. The compiler can then extract the likelihood of a branch being taken from the profile data. Optimizations based on profile data, do not change the semantics and cannot remove code [21, 5].

Some compilers are able to insert *value probes* [2, 11], which can also keep track of specific values. These are currently only emitted for virtual function calls, switch statements, and the size parameters of `mem{cpy,move,set}`. Other generic variables or indirect function calls are not instrumented with this method.

Bird [1], working in the embedded area, has experimented with various sized-focused optimizations on the Linux kernel. He chose to remove the user identifier variables in various kernel structures and replace their usages with zeros, the only possible value if no other users are created. However, the size reduction was not significant, and the run-time performance impact was not measured in the chosen example.

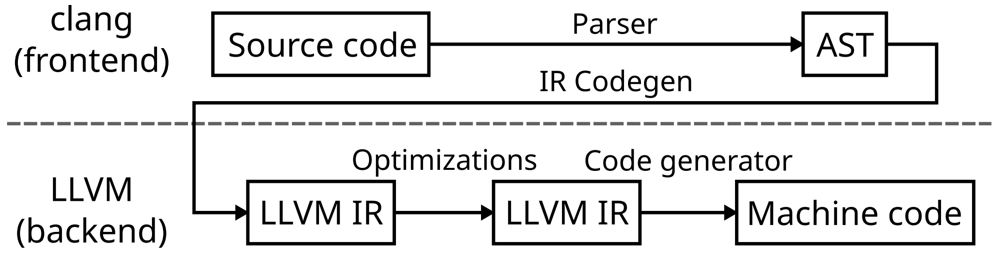

Figure 2.1: The clang compiler only parses and generate LLVM IR, while all important optimizations are done in the LLVM back-end.

We will modify the clang, which is a popular compiler for C family languages. The compiler uses the LLVM compiler infrastructure to generate and optimize code. The overall architecture is shown in Figure 2.1. Clang itself follows a library-oriented design and the parser is usable as a library without having to use the code generation component. The output from the parser and semantic analysis library is an AST of the input file. This allows IDEs to use clang for auto-completion and indexing features.

The generated AST attempts to represent the original source code as close as possible. For example, the parser does not fold simple constant expressions such as $1 + 1$ and instead keeps them intact. This makes it easier to build refactoring tools on top of the clang library.

Listing 2.1 shows a simple C file, that defines a global variable and a getter function. The clang compiler can be instructed to output a pretty-printed representation of the AST. The result for shown C file is shown in Listing 2.2.

```
struct SomeType {
  int x;
  int y;
};
extern struct SomeType gv;

int do_something() {
  return gv.y;
}
```

Listing 2.1: Example C file that accesses a field in a global variable.

The contains all explicitly appearing constructs of the original source code. For instance, the structure with a list of all contained fields. In some cases clang inserts additional AST nodes such as implicit casts. The shown example contains a *lvalue* to *rvalue* cast, which is required for semantic reasons in C and C++. `DeclRefExpr` nodes allow referring to declaration and in our example it refers to the previously defined global variable. Accessing a specific member is represented using a `MemberExpr` node. In both cases the expression also has their corresponding types associated.

```
TranslationUnitDecl
| [...]
|-RecordDecl struct SomeType definition
| |-FieldDecl x 'int'
| `-FieldDecl referenced y 'int'
|-VarDecl used gv 'struct␣SomeType' extern
`-FunctionDecl do_something 'int␣()'
 `-CompoundStmt
  `-ReturnStmt
   `-ImplicitCastExpr 'int' <LValueToRValue>
    `-MemberExpr 'int' lvalue .y
     `-DeclRefExpr 'struct␣SomeType' lvalue Var 'gv'
```

Listing 2.2: The (abbreviated) AST generated by clang for a function that accesses a field in a global variable.

The code generation library traverses the AST to generate LLVM IR. LLVM IR is a machine independent intermediate language that is used by LLVM to represent the code during the compilation. In addition to the in-memory structure, there is also a human-readable form which can be used to inspect the various stages in the compiler. Compared to the underlying processor architectures, the LLVM IR can keep track of the types of values. This allows optimizations to take the actual type in consideration without additional analysis steps. Front-ends such as clang must respect the type system when generating IR. For example, passing a constant integer to a pointer argument in C, would require an explicit `inttoptr` cast in the generated IR.

After the IR has been generated, LLVM can be instructed to run an optimization pipeline on it and generate machine code from the optimized IR. Depending on the optimization level and other flags, it includes optimization such as constant folding, inlining or profile-guided optimizations.

Listing 2.3 shows the LLVM IR for the previous memory load example. While the structure type is still present in the intermediate representation, the representation only contains information about the memory layout. Compared to AST shown in Listing 2.2, the structure type does not contain the name of the fields but only their types. The read from the global variable is represented using a `load` instruction. The address is generated by a `getelementptr` instruction which is used to calculate pointers to elements in aggregate structures. In our example, it calculates the address of the second field in the global variable. The parameters at the tail of the instruction describe the indices to dereference. Because the global variable is of a pointer type, the instruction first dereferences it and then selects the second field. The values are assigned identifiers and local identifiers have a `%` prefix.

```
%struct.SomeType = type { i32, i32 }

@gv = external dso_local global %struct.SomeType, align 4

define dso_local i32 @do_something() {
  %2 = getelementptr inbounds (
    %struct.SomeType, %struct.SomeType* @gv,
    i64 0, i32 1)
  %1 = load i32, i32* %2, align 4
  ret i32 %1
}
```

Listing 2.3: The IR of our simple example is fairly high-level and contains information about the types.

2.3 Debugging Information

High-level programming languages provide many tools for structuring code and data. For example, common structures across programming languages are functions, primitives data types, and simple record-like structure types. When compiling source code to machine code, much of the information about these structures is thrown away. For example, the field names of a structure does not matter for the processor and is therefore unnecessary to keep around. Another scenario where information loss occurs in the translation, is when functions are inlined into other functions. Some parts of the resulting code piece would belong semantically to the caller, but others would belong to the callee.

This circumstance does not pose a problem if the user uses a program and does not care about how the program carries out its work. However, developers may want to know what a program does for debugging or profiling it. Doing so on the machine code itself or even disassembled code would be inconvenient and difficult. The developer would have to first understand what the compiler generated out of the source code and how it maps back.

To encounter this problem, most compilers are able to emit additional debugging data. This includes information about the structures and entities in the original source code. Additionally, it also defines a mapping of entities to machine code or memory representation. Tools such as debuggers or profilers can use the debugging information construct a view of the runtime data that matches the original source code perspective. Commonly used debugging information format are DWARF [7] and PDB (Program Database) [17]. In the following we will give a short overview of the DWARF format.

DWARF is the currently used format for Linux-based systems and toolchains. The central structure of DWARF is a tree of nodes that describes all entities in the program. There are a few other structures such as line number tables for mapping source lines to machine code or abbreviation tables that allows a more compact encoding of the central tree structure.

A node in the main tree structure is called a *Debugging Information Entry* (DIE) and the type of is described by its *tag*. An entry can have a set of attributes describing the node in more detail. An attribute has a name and value, which can have a variety of types depending on the attribute's name and DIE tag. At the top-level there is a DIE for each compilation unit in the program and below that DIEs for top-level definitions. These are separate from each other and do not share types or names for entities. This also means that a type is repeated for each compilation unit it occurs in.

The DWARF standard uses base types to represent primitive types in programming languages. In addition to the base types there are also modifier entries for modifiers commonly found in many programming languages such as `const`, `volatile`, and pointers. These entries refer to the wrapped type and represent the type with the modifier added around it. For the `const int*` type of the C programming language the compiler would emit a base type DIE similar to Listing 2.4. The base type DIE describes the encoding as signed and the size of the `int` type. To describe that the value is `const`, an DIE with the `DW_TAG_const_type` is used that refers to the previously defined `int` base type. Finally, the `const int*` pointer is represented by referring to the previously defined `const int` type with a `DW_TAG_pointer_type` DIE.

```
0x00000192: DW_TAG_base_type
 DW_AT_byte_size (0x04)
 DW_AT_encoding  (DW_ATE_signed)
 DW_AT_name      ("int")

0x00000150: DW_TAG_const_type
 DW_AT_type      (0x00000192 "int")

0x0000018d: DW_TAG_pointer_type
```

```
 DW_AT_byte_size (8)
 DW_AT_type      (0x00000150 "const␣int")
```

Listing 2.4: Description of an `const int*` type in DWARF. The final type is composed using modifiers.

Furthermore, there are tags for describing enumeration, unions, structure and array types. For structures and unions, DWARF includes a list of all fields as children of the structure or union DIE.

Variables are also represented using a DIE and an example for a global variable is shown in Listing 2.5. The DIE contains some source code related information such as filename, line and column number, which is useful for determining where the definition originated from. There is also a specification where the variable's data is stored. For our simple example, it is a single address, but DWARF allows specifying more complex expression to determine the actual location. It also refers to the integer pointer type defined in the previous example, which makes it possible to interpret the data at the storage location.

```
0x000002d0:   DW_TAG_variable
 DW_AT_name        ("int_ptr")
 DW_AT_decl_file   ("/src/code.c")
 DW_AT_decl_line   (40)
 DW_AT_decl_column (1)
 DW_AT_type        (0x0000018d "const␣int*")
 DW_AT_external    (true)
 DW_AT_location    (DW_OP_addr 0xaed0d0)
```

Listing 2.5: DWARF describing an global variable containing a pointer to an integer.

The DIEs of subroutines contain information such as the location of variables, inlined subroutines and subroutine calls. Compared to the global variables, the location descriptions for local variables are usually more complicated. Listing 2.6 shows a variable that has a different location depending on the current instruction pointer. To describe locations in flexible way, DWARF uses a stack-based expression language. In the shown example this is used to describe the value in the first location range as a constant. The first instruction pushes a literal zero on the stack and the second instruction specifies that the value on the top of the stack is the actual value of the variable. If a variable's value is found in a register, as it is the case for the last two ranges, the expression consists of a single `DW_OP_reg` instruction.

```
0x000003b1:     DW_TAG_variable
 DW_AT_name        ("x")
 DW_AT_type        (0x0000018d "int*")
```

```
DW_AT_location        (
  [0x17535f, 0x17536b): DW_OP_lit0, DW_OP_stack_value
  [0x17536b, 0x175377): DW_OP_reg0 RAX
  [0x17537b, 0x1753a0): DW_OP_reg12 R12)
```

Listing 2.6: A variable may be in a different locations depending on the current instruction pointer.

2.4 Memory Tracing

For our work we need to understand what happens in a virtual machine while running a unikernel. Full-system simulation is a tool that allows us to peek into the execution of a guest. Compared to a hardware-based virtualization, the full target system is simulated in software. This also means that the complete processor is simulated in software, which allows tracing every functional detail of the simulation. For example, a memory access by an executed instruction causes the simulator to load a value from the virtual memory. Instead of simply loading the value, the emulator can be extended to log some information about the memory access before continuing the simulation.

Tracing every memory access of a guest operating system comes with its own challenges. The limited available space in registers makes an additional stack structure in the memory necessary. Larger amounts of data are usually stored on the heap region of the main memory. While accessing memory is slower than a register, it is still an operation considered relatively cheap. On most systems there are also cache hierarchies which decrease the latency of such accesses. These factors mean that programs access the main memory very frequently. Tracing such a high-frequency event requires special consideration as it can generate multiple gigabytes per second as raw data. Burtscher et al. have developed VPC4, an algorithm to improve the compression of execution traces [3] The basic idea of the algorithm is based around having a set of predictors. Each time a value has to be processed, every predictor outputs a set of predictions of the value. The index of the first matching prediction is then output instead of the value itself. If no prediction was correct, then the value itself is output. In both cases, the predictors are updated with the actual value and the algorithm is ready for the next iteration.

The output can be then compressed with a general-purpose algorithm such as zstd or LZMA. This setup takes advantage of the greatly reduced entropy in the VPC4 output. Often this allows compressing traces at a much higher compression ratio at the same throughput.

Simutrace uses an implementation of VPC4 to store memory traces in a very compact way [20]. Additionally, it processes the incoming data in parallel to make use of multicore systems.

Chapter 3

Analysis

Unikernels attempt to improve upon appliances based on traditional operating systems by specializing on a single application. For example, they remove any functionality necessary for running multiple applications on the system. Projects such as Unikraft [12] attempt to allow an even more fine-grained specialization. One option is to remove unnecessary components to create a smaller image and another is to replace individual components with more specialized ones to improve the overall performance. Section 2.1 contains a more detailed overview over unikernels. This still requires integrating explicit options which can be enabled or disabled by the developer. In practice, there are two ways to implement this kind of options in C. The first is to use pre-processor `#ifdefs`, which allow inserting code conditionally depending on a condition evaluated during compile-time. The second one is to include all behavior and decide at run-time what behavior should be used. It is usually implemented using conditionals or function pointers in C. Conditionals are usually used for a bounded set of options. A very simple example is a configuration option that enables or disables verbose logging. An example for function pointers can be found at the virtual file system (VFS) of Unikraft or Unix-like systems. Each file system implementation registers a set of operations for FS nodes. This is realized using a structure with function pointers to the actual file system implementation.

Figure 3.1 outlines a simplified version of the `open` function. For file system specific functionality, the function calls the `vop_open` function pointer in the structure provided by the file system implementation. Instead of inserting a call instruction to a fixed address, the compiler inserts an indirect function call. Indirect function calls are usually less efficient, because modern processors with speculative execution has to correctly predict the target to fill the pipeline. They also represent an optimization boundary for the compiler. For instance, for `vop_fallocate` shown in Figure 3.1 both file systems refer to the same no-op implementation. Even when the function is very short and simple, but the compiler cannot inline it as there might be other call targets. This leads to a worse cache usage as the processor cache is usually organized in larger cache lines and the processor has to fetch the whole cache line. Depending on the concrete function that calls `vop_fallocate`, there might be a considerable amount of dead-code code in case all file systems only use a no-op implementation.

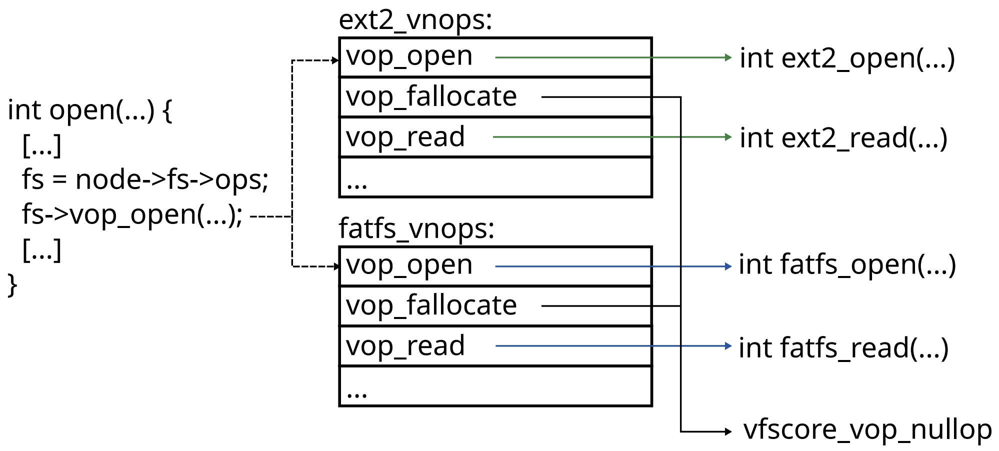

Figure 3.1: Using function pointers is a common abstraction implementation technique. However, the optimizer of the compiler usually cannot assume anything about the values of function pointers.

At some point there has to be a decision what configuration should be active. In case of options that can be configured at run-time, software often reads the desired state from configuration files. These files have a human-readable format and are parsed at the startup of the program. The parser transforms the content of the configuration file to a machine-readable form in memory. Usually, this means that they are stored in structures in global variables or somewhere in a heap allocation. The remaining components of the program can then use these structures to decide on the behavior.

Configuration files in the context of unikernels are often bundled with the appliance itself and do not change after building the image. When the configuration needs to be changed because the requirements change, the image is rebuilt with the new configuration. The resulting new image is then deployed on virtual-machines and the old virtual-machines are stopped. This is in contrast to the classical procedure of changing the configuration file on the file system and then restarting the application process.

The current optimizations in compilers cannot optimize these patterns because there the configuration data is not known at compile-time. In theory, one could replace the configuration file input with a constant string, but the compilers do not evaluate programs far enough to execute whole configuration parsers. Profile-guided optimizations are built in a way that they always preserve correct behavior and are safe even when profiling coverage was not exhaustive. They can move code blocks around or select instructions that are faster in the frequent case, but they cannot remove assume that only specific values occur. In some cases it is possible that variables are known to be constant in a different compilation unit. But the compiler considers each compilation unit as separate from each other by default. Features such as Link-time-optimizations fix this problem by running optimizations across all translation units at the linking phase. This also shares the same limitations of constant expression evaluation that exists classically within a single translation unit.

Our goal is to identify these in-memory configuration structures and collect a list of known-constant values. This list is then converted to a form usable for the compiler. We could create a single dump of the memory state and search for these structures, but would not be able to tell which values change. Creating multiple dumps would suffer from a high storage effort and depending on the dump frequency a low coverage. Instead, we can create a trace of all individual memory accesses which allows us to watch for changes and reconstruct the memory image if necessary. For the identification we need determine some robust properties to classify memory locations. In the following we will assume that we can create a trace of the unikernel that is complete in the sense that all code-paths are exercised. We can observe that the values of these configurations do not change after the initialization phase of the unikernel. This is not enough information for retrieving the actual values as it is not clear when the initialization ends. If we instrument the unikernel to indicate the end of the initialization phase, then we could copy the configuration data at that point. The downside is that the developer is responsible for identifying the configuration structures and would have to instrument all applications.

Another way is to look at all written values to a memory location. If only a single value is observed during the lifetime of the unikernel, then the compiler could have assumed that the value was actually a constant. One problem with this approach is that it would break if for example the structures are zero-initialized using the `memset` function. When looking at the written values there would be two values: the desired value and a zero.

We can fix this problem by looking at read accesses instead of writes. These represent the values that are actually observed by the program. Concrete values that were written but never read were never observed and cannot have an effect on the execution. However, there will be still cases where multiple different values are observed during the initialization phase. For example, the configuration structures might have a size field that increases as more of the configuration file is parsed. With our approach this field will have multiple values associated with it, because increasing the value also incurs a memory read.

We decided to track all data at byte-granularity and not at base-type-granularity. The latter is strictly more descriptive than the former but is more complex to implement. For example, when tracking a four-byte integer at base-type level we could distinguish two values `0x00000000` and `0xFFFFFFFF`. In the case of byte-level tracking there would be two values for each byte resulting in $2^4 = 16$ indistinguishable combinations. For our work we only care about unique values and the value is already considered non-constant when there are at least two unique observed values. Implementing the more detailed base-type-granularity would not have any benefit for our work.

Further, we decided to track the values based on the address or correspondingly to heap-allocations the offset within the encompassing type. One could also do this analysis for each memory access instruction separately. While it could potentially provide a more fine-grained result, it would also increase the risk an insufficient value-coverage. This is the case because every value also has to additionally be observed at every read location instead of only at one location. But it could have benefits in situations where multiple values are observed during the initialization phase and only a single during the remaining execution.

We chose to implement the memory tracing from the outside in the full system simulator. Implementing the memory tracing by instrumenting unikernel itself would have several downsides. In theory, it would be possible to modify the compiler to include a functional call for every memory access in the code. However, this would considerably increase the code size and substantially changing the code layout compared to a non-instrumented build. Another problem would be that our approach depends on having all memory accesses. A common optimization is to use hand-crafted assembly implementations for frequently used functions such as `memcpy`. These implementations would not be instrumented with a compiler modification and would need to be instrumented by hand. Finally, the resulting data has to be transported outside the virtual machine and stored on the host. The transport mechanism would have to be excluded and also be efficient enough to slow down the execution to unusable performance. In comparison, implementing this in full system simulator is rather simple. A full system simulator has to translate memory access of the guest code to some equivalent for the host. At the point where this happens we can insert an additional function call that traces the address and value of the operand. The implementation can also benefit from the virtual memory management of the host and directly process the resulting data.

One limitation of our approach is that we rely on seeing all values in the profiling run, that will also show up in real-world conditions. In many cases it is hard or impossible to ensure this in the general case as having a witness of a value does not prove that other values are not possible. In the optimal case a developer might try to record a unikernel in real-world conditions. However, they still would have to decide when enough data was recorded for useful dataset and stop the recording. This is not unlike a central problem in software testing as described by the famous quote of Edsger W. Dijkstra: “Testing shows the presence, not the absence of bugs”. In practice, the developer can add additional guidance to the tooling to restrict it to known safe structures. For example, they can provide a list of configuration structures which are known to be constant after the program initialization. If there are other constants that were detected, they can be output for further analysis.

We decided to store the resulting trace data instead of processing it directly. A benefit of this is that we can incrementally develop the analysis program and refine the functionality repeatedly on the same data. Because the data is well compressible as described in Section 2.4, the storage overhead is manageable. Processing it synchronously would also slow down the simulation substantially and impact the interaction with tools such as HTTP clients outside the virtual machines because of timeouts.

For the analysis we need to know the virtual memory layout within the guest at each point in time. The unikernel file already contains information about the basic memory layout including code and data segments. But notably this does not include any information about the heap, which is managed completely at runtime. We want to keep the amount of necessary modifications to the unikernel to a minimum. While we could parse the heap allocator structures to find out the exact contents of the heap, this makes in dependent on the specific allocator. We decided to instrument the allocator interface to create a trace of allocation, because it is independent of the used allocators and less error-prone because we do not have to re-implement parts of the allocator.

After the analysis we want a list of all detected constants in a machine-readable form. Currently, compilers have no mechanisms to inject values at specified locations. Therefore, we need to implement such a mechanism on our own. Specifically, we want to replace references to fields within structures with constants. But other possible actions would be to remove unused fields from structures.

Chapter 4

Design

Based on the findings in the previous chapter, we want to design an approach to recognize run-time constants in a unikernel and inject them into the compilation phase. Section 4.1 describes how we want to gather the desired constants based on the run-time behavior of the unikernel. In Section 4.2 we describe how we plan to integrate the constants into the compilation process.

4.1 Data Interpretation

As described in Section 3 we want to find out configuration options and variables stay constant during the execution of the program. To find out how the data is structured at run-time we want to make use of debugging information. We described in Section 2.3 how the DWARF debugging format allows external programs to inspect the state of a program. Most importantly, they include a description of the variables in a program and their types. These debugging information formats are mostly designed with debuggers and profilers in mind. Debuggers allow the user to inspect the program state from the viewpoint of the original high-level language. A common task in this scenario is for example to print the value of local variable. In that case, the debugger can retrieve the current location of the variable from the debug information.

We need to figure out what type is used to access the data within a heap allocation. In the C programming language the allocator interfaces do not handle any type information. The program only passes them a requested size and if necessary a minimum alignment requirement. The resulting pointer is then cast to the desired type and accessed via that type. In some applications the system allocator is abstracted away through potentially many abstraction layers and at a much later point in the execution. This makes it much harder analyze the debugging information statically and determine a type reliably.

Therefore, we collect the instruction pointers of the memory access instructions. We can then analyze the instructions to find out the variable that was used to store the pointer. Depending on how the memory was addressed we try to figure out which variable was used to store the original pointer to a structure. We want to do this using the debugging information emitted by the compiler. By traversing this debugging information we can try to figure out what variable is likely used to store the pointer used for the memory access. For global variables we can also figure out the exact type with the help of the debugging information. They store the exact address of the variable in the binary and its type.

For each memory address we can store the observed values and group them into the respective base types. With the resulting information we can determine which parts of a type are always constant.

4.2 Compiler

We described the organization of a modern compiler in Section 2.2. Compilers are roughly separated into a front and back-end. The clang compiler is considered a front-end for C programming languages and uses the LLVM compiler infrastructure its back-end. Based on the arguments outlined in Section 3, we decided to modify the front-end to carry out our optimization. The LLVM IR does not carry enough information to reliably select the correct constant injection points. Therefore, we implement our modification in the clang front-end.

While parsing a source code file, clang builds an AST representation in-memory. The compiler then walks through this tree by function and generates LLVM IR for each one. We want to replace member accesses using a memory load with a constant value. There are multiple possible points to accomplish this goal. We could add an optimization in the back-end as an LLVM optimization pass. This would allow easier reuse of our optimization across other programming language with compilers which use LLVM as their back-end. While there is some level of type information available in the LLVM IR form, some high-level concepts such as field names are not part of the representation. Another possibility is to implement the optimization as a source-to-source transformation. In theory, such program would read the unikernel source-code and replace all references to the desired variable with constant literals. A simple search and replace approach would be very fragile as it would difficult to cover all usages. For example, for the expression `v->x` it is not clear what type `v` has without parsing the context. In the ideal case, such source-to-source transformation tools would have access to the parsed AST and semantic information such as types about the code. The clang compiler is organized in a way that makes writing such tools possible without having to implement a complete compiler front-end. Additionally, clang provides various utilities to simplify writing tools such as `LibASTMatchers`. The clang AST provides source code location information which can be used to replace existing text. For example `memberExpr(member(hasName("uid")))` would allow matching against all member expression where the field name is *uid*. In our case we could match against `memberExpr()` to match against *all* member expressions and match them against the list of detected constants. We could replace all desired expressions with constants using the source code location information and write the resulting contents to disk.

However, the normal clang compiler driver also uses a similar pattern: it parses the source code and then traverses the AST to generate the code for each function. Instead of having an intermediate stage as in the source-to-source transformation, we can directly apply our adjustments to the LLVM IR generation step. This results in a cleaner integration into the existing build system and allows enabling the optimization by passing an additional compiler flag. Therefore, we decided to modify the compiler itself to carry out the optimization.

A conceptual limit is that pointers to fields are not covered by this approach. For example, a structure might contain an integer, and some function might create a pointer to the field. Any access via this pointer could not be associated to the structure field. Therefore, we cannot replace such loads with constants.

As discussed previously, a risk of our approach is that there is no safe criteria when all possible values are covered. In some applications, this could cause the unikernel to misbehave or even create security issues. A possible solution would be to integrate an optional verification mode. For every replaced load this mode could insert an assertion that ensures the value in memory matches the expected constant. If the value does not match, the program would abort instead of continuing with incorrect behavior.

There are also fields are never read during the unikernel execution. For example, there may be fields that are only used by optional features which were not enabled. This can happen for both features that were disabled at compile-time and features that were never used due to run-time behavior. In the first case, there are no field accesses at all, and in the latter case the compiler might still emit loads which were not executed during run-time. The fields may also be never read because our optimizations removed all reads. In these cases, the field still takes up space in memory, but is never really used anymore. A possible optimization could remove the unused field altogether and reduce the size of the affected structures. To handle any remaining loads that are expected to be dead-code, we can replace them with undefined or poison values. Replacing them with undefined values would increase the freedom of the compiler to optimize or even remove the code after the affected load. The verification mode could instead insert an abort function call, which stops further execution.

Removing stores to detected constant fields requires more consideration. Our current approach only replaces direct references to fields in structures. Listing 4.1 shows a case where our approach relies on having the (constant) value in memory. The code creates a pointer to the integer field `x` in `extract` and accesses the field via this pointer in `extract_field`. The compiler only sees a dereference operation of an integer pointer without knowing where this integer is stored. If we still emit all stores, the value is still stored in memory and the load in `extract_field` still returns the correct value. In C there is no way to find out the enclosing type given a pointer to a field. Therefore, we have to emit stores to fields which are potentially referred by a pointer. We can create a list all fields that are used in an address of operator or `offsetof` macro. Stores to fields in this list should be still emitted, but known reads can still be replaced.

```
struct st {
  int x;
}
int extract_field(int *ptr) {
  return *ptr;
}
int extract(struct st *s) {
  return extract_field(&s->x);
}
```

Listing 4.1: C allows creating addresses to fields and code using those pointers does not know the surrounding type.

In Chapter 3, we identified function pointers as a possible optimization barrier. Because the compiler does not know what set of functions could be called, it does not have the option to inline the function bodies of the called function. This prevents further optimizations such as constant propagation or folding. The analysis approach described, also works for function pointers as indirect calls also cause a memory read. The values collected this way describe pointers to the code section instead of plain values. The concrete values depend on the location where a function is stored in memory. All functions in a binary are laid out sequentially, which means that changing the code size of a function also moves all functions afterwards around. The expected effect of our optimization is that we reduce the code size, and therefore we cannot directly use the raw pointer values. Instead, we would have to insert a symbol reference and rely on the linker to insert the correct addresses. Many functions used for filling function pointers in implementation abstraction structures as described in Chapter 3 have internal linkage. This means that the symbol of the target function is only visible within the compilation unit and not in other compilation units. In some build system setups such as the one used by Unikraft, there is a step that changes the linkage to internal after all compilation units within a component have been linked together. For symbols intended to be used by other components there is a list exported symbols, which retain their external linkage.

The first problem of being unable to reference internally linked symbols, could be solved by introducing an additional unique symbol that points to the same address as the normal symbol. Choosing unique names is important because the symbols component-internal functions normally do not conflict with any other symbol and therefore do not need to be unique across all source files. With this approach any non-unique symbols would cause a conflict if the externally-visible names only depend on the old identifiers. The second problem could be fixed by modifying the build system to not touch these special symbols.

Chapter 5

Implementation

Based on the design outlined in Section 4, we describe our implementation in the following. We first describe how we gather the raw data about the execution of a unikernel in Section 5.1. Section 5.2 explains how we process this raw data and reconstruct data structures from it. Finally, Section 5.3 describes the implementation of our optimization in the compiler itself.

5.1 Data Acquirement

The hypercall necessary for collecting information about the heap is implemented by reading from an unused model-specific-register on x86 platforms. The necessary code for the QEMU and guest is generated by the scripts part of Simutrace [20]. Only slight modifications were necessary because the guest code was written with the Linux kernel in mind. To cross-check the gathered data and analysis results we also implemented the collection of a stack-trace. This allows us to check the code which triggered the allocation and verify that we figured out the correct type.

The QEMU modifications also include a trace point for memory access by the guest. The generated event includes the virtual address, memory value, access size, and instruction pointer value. It also includes the current value of the cycle counter, but we did not use it for our work.

All events are packed into separate packets which are then compressed. For the memory events, the individual fields are processed using the VPC4 algorithm and then compressed using zstd. The remaining allocation tracing events are compressed using zstd.

5.2 Analysis Tool

The next step is to analyze the traces generated by QEMU and output the constants for the compiler optimization. We wrote the analysis tool in the Rust programming language as it provides a modern environment to implement the analysis tool. We chose Rust because there are mature libraries for the ELF and DWARF formats, which allow working with them in a mostly type-safe way. Because of the involved amounts of raw data choosing a compiled language was essential for us.

The basic operation of our tool is to open the traces and iterate over all events in the contained data streams. Simutrace takes care of decompressing the streams packet-by-packet and applying VPC4 in reverse for the memory events. The result of this operation is the reconstructed data of the packet. We can then directly access the individual events in the packet by accessing it using a `struct` matching the structure of the C code.

We use the allocation events to keep track of the heap state. For every allocation event we store an entry in a B-tree map with the allocation base address as the key. The value of the map entry consists of the allocation size and the call-stack of the allocation call. When the unikernel deallocated some memory, we can find the correct heap entry via the base-address. The B-tree map allows us to quickly search for the correct allocation that contains a given address.

A small but important data structure is the `ValueList`, which represents a single byte in memory. The structure tracks up to five different observed values instead of up to the 256 theoretical possible values. While we only need to know whether there was a single unique value for the final compiler guide, we track multiple values for informative purposes. The structure also tracks all instruction pointers that accesses the memory location. This list will be used to determine the type of the heap allocation. To quantify the importance of a structure the `ValueList` includes the read and write access count.

As previously described, we track all instruction pointer that access a memory address. The instruction pointer allows us to retrieve the instruction that caused the memory access. We can then disassemble the instruction and inspect it. We then look at the operands of these instructions and check which segment was accessed. In case a non-data segment such as the stack was accessed we ignore the instruction and continue with the next one. For data segment accesses we check whether there is a base register for the address calculation. In most cases the compiler emits a field read in a structure with a base and displacement addressing mode. The base is usually the start address of the structure and the displacement is the offset within the structure to the desired field. As previously described, the debug information for a function also includes the location of a variable. We can scan through function that contains the considered instruction and look for a variable that is stored in the base register. If there is a variable that satisfies this condition, we can use the type specified for the variable. For some instruction pointers the resulting type is not useful as the function uses a generic type to access the memory. An example for this is when a function zeros the structure using `memset` because it usually uses the `char` type to write zeros to the memory range. Based on the symbol table of the unikernel we generate ranges of blacklisted functions and if an instruction pointer falls within such range we ignore that instruction pointer.

The next step is to merge all instances of a type together and create a summary of the different values across all instances. Because of the compilation model of the C programming language is centered around translation units, the compiler generates the debug information for each one separately. In case of functions this is a sensible model because each function is unique. But for types which are usually defined in headers this means that a type appears in the debugging information of multiple translation units. When we figure out the type by looking at the debugging info of a function, the variable DIE points to a type within the compilation unit. If a function in another compilation unit exists that accesses the same allocation, the variable DIE refers to a different place. This means that a simple identity comparison is not enough to merge the instances of an individual type. Instead, we have to use a structural comparison operation to decide whether a type is the same. Because we store all types in a hash map indexed by the type itself, the same applies for the hash function.

To implement the comparison and hashing operations, we use a recursive strategy. Two DWARF base types are equivalent iff the bit size and their encoding are the same. For structures we consider them equivalent iff the members excluding member names are equivalent. The type modifiers such as `const`, `volatile`, pointer are equivalent iff the modifier is the same and the inner type is equivalent. A similar pattern is used for the hashing implementation. Instead of comparing the specific property with another type's property, the value is added to the hash. The recursion adds the inner type to the hash, resulting in a hash over the whole type.

A problem with a naive implementation of this approach is that C types can have cycles in their structure. Linked lists are a well-known example of this pattern and Listing 5.1 shows a simple linked list structure.

```c
struct list_node {
  struct list_node *next;
  int value;
};
```

Listing 5.1: Linked lists are a popular data structure in C and the type contains a cyclic reference to itself.

Each linked list node refers to the next node and usually the `next` pointer of the last node is set to the `NULL` pointer. The type of the `next` member refers to the type that is going to be defined. A comparison implementation not considering this fact, would endlessly traverse into the type itself. To fix this, our implementation keeps track of the types that are compared along the recursion. When we recurse down, we add the current comparison to the list of active comparisons. After finishing a comparison-step of a type we remove the comparison again. Every time the algorithm traverses down it also checks whether there is the same comparison at a lower recursion level by checking this list. If there is such an active comparison, then we consider the types as equivalent on that level. The only way such a recursive type is non-equivalent is that there is at least one non-matching comparison outside these explicit break-points.

Another problem stems from the fact that C has the concept of incomplete structure types. They do not have a list of members associated, but it is already usable as a pointee. This is sometimes used to hide the structure members in the public interface of a library. In that case the actual structure is only defined in the actual implementation source file and all interfaces only pass around pointers to the structure. For client source files that include the public interface header these structures show up as declaration-only types in the debugging information. This makes them impossible to compare them structurally across multiple translation units. For structures affected by this we instead rely on the structure name and assume that naming conflicts are unlikely to exist.

In some cases the analysis can end up with multiple types for the same allocation. This situation can arise if there is a pointer to a sub-structure in the allocation. If some code accesses the memory of the allocation with such a pointer, then this type will also be a valid type for a portion of the allocated memory. The tracing data does not contain enough information to figure out the precise location of these sub-structures in all cases, because x86 supports addressing modes such as Base $+$ (Index $*$ Scale) $+$ Displacement. In the case of the mentioned addressing mode, both the base and index are registers. The other variables are constants encoded in the instruction itself. Because only the final address is known and there are two unknown variables in this equation, we cannot determine the value of base in the general case. For simpler addressing modes with only a single variable, we could calculate the base address and puzzle together the overall type of the allocation. However, we decided to implement an algorithm based on a simple observation: when structures are composed of other types, then the encompassing type must be at least as large as the inner type. Therefore, the algorithm can choose the largest detected type for the allocation.

For memory accesses that do not fall into an allocation but instead into the data segment of the unikernel, we accumulate them until of the trace end. After we iterated through all events, we associate all accessed addresses to global variables. We can find a description of global variables in the DWARF debugging information. They are represented as variable DIEs directly below the compilation unit DIE and contain the address range where they are stored. By comparing the ranges in the debugging information and the actual addresses we can merge the values into the previously described type-value mapping.

The accumulated data is then output as an HTML report or YAML report for the compiler. The HTML report contains all detected types and a summary of the observed types. This report includes a detailed dump broken down to the bytes on a base type level. For each byte, the report shows whether there were no values, a few different values, or too many observed. For all types it also includes the read and write count to give a rough importance indicator.

The YAML report for the compiler contains a similar set of information. But compared to the HTML report we filter out fields with more than one value.

5.3 Compiler

As discussed in Section 4.2, we decided to implement our optimization directly in the compiler front-end itself. After the compiler has parsed the source code, it iterates through all functions in the source code and generate LLVM IR. On a statement and expression level this happens using a visitor pattern for the AST nodes. For our optimization, we are interested in the member expression nodes that are translated to memory load instructions. This happens in the `ScalarExprEmitter::VisitMemberExpr` method. This method first attempts to evaluate the referenced field as a constant. For example, the structure might be stored in a local `const` variable and therefore the compiler can figure out the value. If this is not possible, a memory load from the address of the field is emitted.

We inserted a check between these two cases that compares the path of the member expression with the paths specified in the YAML file from the analysis tool. This check also considers nested `MemberExpr` nodes in case of nested structures. When the YAML file contains the considered field, we construct a value with the content specified in the file. LLVM uses arbitrary precision integers for representing the values, and we start by constructing such value. The memory and value representation does not always match. For example, a boolean is represented by a single byte in memory, but are represented as a single bit type in the LLVM IR. Therefore, the next step is to convert the memory representation to the correct type in the LLVM IR. We return the resulting constant instead of the IR value or the memory load.

Chapter 6

Evaluation

In the previous chapters, we described the design and implementation of an optimization that allows injecting constants into a unikernel. The constants are determined based on a memory trace of the unikernel itself. In this chapter, we want to evaluate the effect of applying the optimization and find out whether it improves the performance or size of unikernels.

6.1 Methodology

To evaluate the effect of the implemented optimization, we want to measure how our optimization affects the size of the built unikernels. For this we take a look at the size of the ELF sections and compare them between unikernels built with and without our constant propagation optimization. In particular, we want to focus on the `.text` and `.data` sections in the image. The `.text` section contains the machine code and our optimization should result in less code as outlined in Section 3. The `.data` section can shrink when references to contained variables are replaced by constants. We also check whether the resulting unikernel still works correctly.

Figure 6.1 shows an overview of our evaluation process. To gather the necessary information for our analysis tool, we build each unikernel with debugging symbols. We disable any optimizations to prevent the compiler from removing memory accesses which our tool needs to determine constants. The resulting unikernel is then run within the modified QEMU with memory tracing support. In case of server applications, we apply external inputs such as HTTP requests to the running server. The memory trace is then analyzed by our tool, which outputs an HTML report and a YAML file. In some cases, we have to filter out some detected constants, because the values depend on the specific compiler output. We build the final unikernel with optimizations enabled and enable our optimization by passing the YAML file to the compiler.

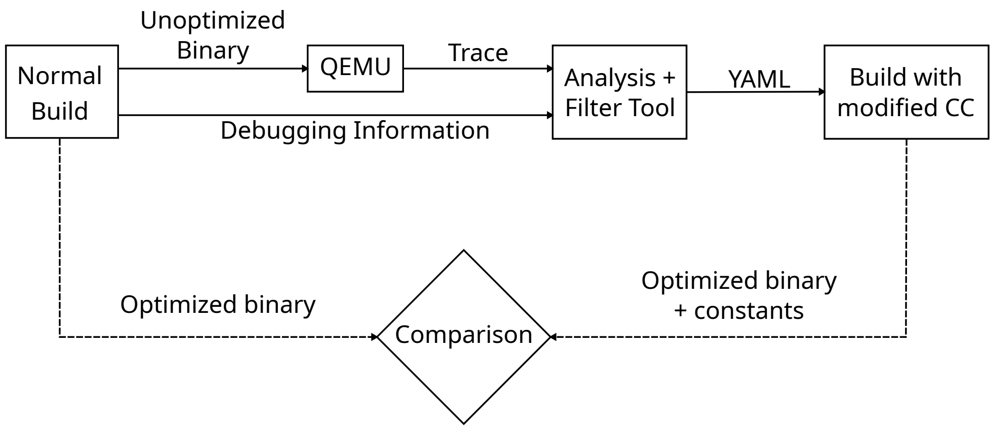

Figure 6.1: We compare the binaries produced by a normal build with those produced by our optimization pipeline.

The hardware and software configuration, which was used for our measurements, is shown in Table 6.1. We used GCC instead of clang for binaries with debugging information, because GCC tended to produce more reliable results.

Component	**Specifications**
Processor	AMD Ryzen 7 PRO 5850U
Main Memory	32 GiB
Storage	Samsung PM981a, 1 TiB
Guest Main Memory	500 MiB
Software	**Version**
Host Kernel	Linux 5.18.10
Guest	Unikraft 0.6 (Dione), including modifications for PHP
Compiler	GCC 12.1.0 (debuginfo binaries), LLVM/clang 15.0.0-f69328049e9e with modifications (constant injection)
QEMU	2.6.50, modified for Simutrace
Simutrace	4.0

Table 6.1: The hardware and software used to evaluate our implementation.

6.2 Microbenchmarks

Our first micro-benchmark represents a minimal unikernel to test how detecting run-time constants can further eliminate dead code. The full unikernel is shown in Listing 6.1. It takes a parameter from the command line and outputs a message if the parameter was not 12. Internally, it converts the parameter into an integer and stores the value into a configuration structure. This structure resembles a typical way of holding configuration parameters. For this unikernel, there is one other function that reads the configuration from the structure and outputs a message depending on the value.

```
#include <stdio.h>
#include <stdlib.h>

struct {
        int arg;
} config;

void __attribute__ ((noinline)) print_argument(void) {
        if (config.arg != 12)
                printf("hello␣world:␣argument␣is␣%d\n",
                       config.arg);
}

int main(int argc, char **argv) {
        if (argc != 2)
                UK_CRASH("invalid␣number␣of␣arguments");
        config.arg = atoi(argv[1]);
        print_argument();
        return 0;
}
```

Listing 6.1: The simple demonstration unikernel parses the parameters and outputs some text depending on the value. The exact run-time behavior is not known at compile-time and depends on the aforementioned parameter.

Suppose the parameter is always set to “12” and therefore the configuration value is also always the same. This is comparable to programs that parse configuration files from immutable disk images at startup. Because the value is only known at run-time, the compiler cannot remove the conditional in `print_argument()`. The resulting machine code for the function is shown in Listing 6.2 and confirms that the compiler inserted a check for the current value of the argument at line 8.

```
1 000000000010a430 <print_argument>:
2   ;; Function preamble
```

```
  10a430:       push   rbp
  10a431:       mov    rbp,rsp
  ;; Load config.arg
  10a434:       mov    esi,DWORD PTR [rip+0x1c82e]
  ;; Compare with constant 12
  10a43a:       cmp    esi,0xc
  10a43d:       jne    10a441 <print_argument+0x11>
  ;; False branch
  10a43f:       pop    rbp
  10a440:       ret
  ;; True branch
  10a441:       lea    rdi,[rip+0x144d1]
  10a448:       xor    eax,eax
  ;; Tail-call to printf
  10a44a:       pop    rbp
  10a44b:       jmp    10b790 <printf>
```

Listing 6.2: The compiler has to emit a memory load for the configuration variable and has to include code for all possible values of the configuration parameter.

We generated a memory trace as described in previous chapters and analyzed it with our tool. The memory trace for this unikernel is 156 MiB of raw data describing 1.145.910 memory accesses, which ends up compressed as 1.2 MiB. The analysis tool detected nine constants, which are shown in Table 6.2. Most of the accesses to detected constants can be attributed to the console components of Unikraft. These constants only make up a very small portion of the data read from memory.

We then passed these constants to the modified compiler and rebuilt the unikernel. The resulting assembly code for the `print_argument` function is shown in Listing 6.3. Because all reads from the `config.arg` field were replaced by a constant "12", the condition is trivially false. This allows the dead-code analysis to remove the complete if-clause resulting in an empty function.

```
000000000010a180 <print_argument>:
  10a180:       push   rbp
  10a181:       mov    rbp,rsp
  10a184:       pop    rbp
  10a185:       ret
```

Listing 6.3: The injection of the constant enabled the compiler to evaluate the if-condition and remove it completely.

The text section of the unikernel without our optimization is 127051 bytes. With the cleaned up version of the constant list the binaries text section shrunk to 126091 bytes. Therefore, applying our optimization results in a 960 bytes smaller

Type	Path	Reads	Description
variable	plat_allocator	1	Allocator used by platform code
variable	virtio_drvs.tqh_first	1	List of virtio devices
type	virtio_driver_list.tqh_first	1	List of virtio devices
variable	config.arg	1	Configuration parameter
type	<unnamed>.arg	1	Configuration parameter
variable	tsc_mult	2	Multiplier for x86 TSC ticks to nanoseconds
variable	terminal_color	239	Default terminal color
variable	dreg	492	VGA data register
variable	areg	984	VGA address register
Total		1722	

Table 6.2: The list of detected constants contains our configuration variable among other platform related constants.

text section.

In many cases the configuration is not placed in a global variable but stored in a heap allocation. We modified our simple unikernel to replicate this pattern and Listing 6.4 shows the source code. The configuration structure is allocated in the `main` function, filled with the parsed value and then passed as an argument to the `print_argument` function. The `print_argument` accesses the configuration via this pointer argument.

```
#include <stdio.h>
#include <stdlib.h>

struct config {
        int arg;
};

void __attribute__ ((noinline)) print_argument(
        struct config *cfg
)
{
        if (cfg->arg != 12)
                printf("hello␣world:␣argument␣is␣%d\n",
                        cfg->arg);
}

```

```
int main(int argc, char **argv)
{
        if (argc != 2)
                UK_CRASH("invalid␣number␣of␣arguments");
        struct config *cfg = malloc(sizeof(struct config));
        cfg->arg = atoi(argv[1]);
        print_argument(cfg);
        return 0;
}
```

Listing 6.4: We modified the unikernel to store the configuration on the heap instead of a global variable.

We traced this unikernel in the same fashion as the variant with global variables. The analysis results match the ones described previously and also correctly identify the constant in the configuration structure. The same applies for the results after injecting the constants into the compilation process and the compiler is able to remove the complete if-condition.

6.2.1 Discussion

Part of the detected constants is also a structure describing the memory layout of the virtual machine. The tool correctly detected that these values did not change for the considered unikernel and (same) virtual machine memory allocation. Changing the contents of the unikernel binary by removing code will also slightly change the memory layout. In this case, there will be more room for the heap, because there is less code after we apply our optimization. This causes the start addresses and lengths of the segments to change. On x86 the structure is used to store the regions of the boot stack, initial-ram-disk, and heap. In this example, there is no initial-ram-disk and there only a few allocations in the heap. This increases the likelihood that the unikernel is functional even if the regions are incorrect. The previous unikernel confirms that this is the case and still works from a functional point-of-view. We excluded the structure for our evaluation because it breaks completely once an initial-ram-disk is loaded and there is an unnecessary risk of introducing subtle errors even if the unikernel works at first sight.

6.3 Macrobenchmarks

To evaluate the impact of our optimization on a larger application, we used a web server unikernel. This appliance is based on the nginx 1.21.4 web server and includes PHP 8.1.0 for server-side scripting. The combination is widely deployed across the internet and represents an interesting opportunity for this optimization as the configurations only rarely changes once deployed. The configuration files are stored on an initial-ram-disk and the files served are stored on an ext2 disk image. For `.php` files nginx forwards the requests to the PHP interpreter which runs the requested script and returns the output to be sent back to the client. As part of the tracing phase, we sent two requests to each of three endpoints: a simple HTML file, the classical `phpinfo()` script, and minixed [24], which is a simple PHP directory browser application. The tracing phase generated 5.2 GiB of raw data (102 MiB compressed) containing about 59 million memory reads and 35 million memory writes.

For this workload, the analysis tool was able to generate a list of 1623 values that it determined to be constant. Starting the unikernel with all constants does not work because of the aforementioned memory layout variables. While the unikernel boots after removing these constants, it is not able to respond to any HTTP requests. Upon closer inspection the network stack was stuck assigning a unique index for the external network interface. This happened because the analysis script determined that the index field of network interfaces would always be zero. The loop that repeatedly incremented the network interface number until does not match one of an existing interface would therefore never terminate. Removing the constant did allow the network stack proceed normally but the unikernel did still not answer to HTTP requests. However, the network stack can respond to ICMP ping packets, which suggests that a constant on the application layer breaks the unikernel. To determine which constants result in a non-working unikernel, we wrote a script that recompiled the unikernel with each type-based constant injected individually. For each compiled unikernel the script checked whether it was able to respond to an HTTP request. Table 6.3 lists all constants that had an impact on the binary size and whether the unikernel booted with the constant injected. After determining which constants cause a breakage, we generated a list of all working constants and compiled the unikernel with this list. The resulting unikernel crashed on the first request to an PHP file. On closer inspection, the constant for `fd` field in the `ngx_open_file_s` structure causes nginx to erroneously write into a `epoll` file descriptor. Removing this constant from the set fixes the problem and the unikernel is able to respond to all requests.

In addition to the type-based constants, we also verified that only including constants detected in global variables results in a working unikernel. Including those also reduced the code size by almost 5 KiB. Finally, we combined the set of global variable constants and type-based constants, which reduced the code size by about 12 KiB.

Target	`.text` **size**	Δ `.text`	**Works?**
(Reference without optimization)	15208839	0	✓
(All detected constants)	15181191	-27648	✗
(All variable constants and passing type-based constants except ngx_open_file_s.fd)	15196103	-12736	✓
vnode.v_type	15196295	-12544	✗
(All passing type-based constants)	15201095	-7744	✗
(All variable constants)	15203847	-4992	✓
_zend_object_handlers.do_operation	15206919	-1920	✓
_zend_array.pDestructor	15207367	-1472	✓
_zend_module_entry.type	15207687	-1152	✓
vnode.v_refcnt	15207943	-896	✓
_cwd_state.cwd_length	15208071	-768	✗
netif.num	15208391	-448	✗
vfscore_file.f_vfs_flags	15208519	-320	✗
_reent._atexit	15208583	-256	✓
netif.next	15208583	-256	✓
vfscore_file.f_flags	15208583	-256	✓
_zend_llist.persistent	15208647	-192	✓
netif_ext_callback.next	15208647	-192	✓
_zend_module_entry.zend_api	15208647	-192	✓
driver.devsz	15208711	-128	✓
_zend_array.nInternalPointer	15208711	-128	✓
_zend_module_entry.module_started	15208711	-128	✓
vfscore_file.f_offset	15208711	-128	✗
posix_socket_ops.init	15208775	-64	✓
_xmlSAXHandlerV1.initialized	15208775	-64	✓
posix_socket_driver._list.tqe_next	15208775	-64	✓
netif.status_callback	15208775	-64	✗
_zend_internal_function.fn_flags	15208775	-64	✓
pthread_attr_t_.inheritsched	15208775	-64	✓
curl_version_info_data.age	15208775	-64	✓
curl_version_info_data.libssh_version	15208775	-64	✓
_zend_module_entry.handle	15208775	-64	✓

Table 6.3: Only a few constants have an impact on the binary size and some of them also result in a broken unikernel.

6.3.1 Discussion

We described in the previous chapters how we to reconstruct the types of an individual heap allocation. When taking a closer look at the failures in the nginx appliance, we can observe that the constant detection for global variables works reliably. However, some constants detected in heap allocations break the unikernel. The analysis tool outputs a list of heap allocations where it was unable to associate a type to. This includes the list of instruction pointers where the memory in the allocation was accessed. Listing 6.5 shows the disassembly of one of these locations. The code is part of a generic reference counter utility in Unikraft and is responsible for initializing the reference count. The memory access to the heap allocation is in line 5 and the instruction uses the address stored in the `rax` register. This address was read from the stack two instructions earlier in line 3. In this case the analysis tool looks for the DIE of the `uk_refcount_init` function and searches for a variable that is stored in the `rax` register.

```
1 0000000000182be8 <uk_refcount_init>:
2   ...
3   ; ukarch_store_n(&ref->counter, value);
4   182c85:    mov    rax,QWORD PTR [rbp-0x18]
5   182c89:    mov    edx,DWORD PTR [rbp-0x1c]
6   182c8c:    xchg   DWORD PTR [rax],edx
7   ...
```

Listing 6.5: The compiler loads the necessary address from the stack right before it is used instead of keeping it around in a register.

Listing 6.6 shows the debugging information for the function, which notably does not include any variable in a register. Instead, all variables and parameters are stored on the stack. By analyzing the assembly code we know that the address to the heap allocation was loaded in line 3 and was stored on the stack with an offset 24 from the base pointer. This corresponds to the DIE at line 20 which describes that the variable `_r` is stored at that location. The correct conclusion would be that a part of the allocation is of the type `__regs`.

```
1  0x0000fb7a:   DW_TAG_subprogram
2                  DW_AT_name      ("uk_refcount_init")
3                  DW_AT_low_pc    (0x182a96)
4                  DW_AT_high_pc   (0x182b3f)
5
6  0x0000fb98:     DW_TAG_formal_parameter
7                    DW_AT_name    ("ref")
8                    DW_AT_type    (0xfbd7 "__atomic *")
9                    DW_AT_location        (DW_OP_fbreg -40)
10
```

```
0x0000fba7:     DW_TAG_formal_parameter
                  DW_AT_name    ("value")
                  DW_AT_type    (0xc251 "__u32")
                  DW_AT_location        (DW_OP_fbreg -44)

0x0000fbb6:     DW_TAG_lexical_block
                  DW_AT_low_pc  (0x182ab5)
                  DW_AT_high_pc (0x182b33)

0x0000fbc7:       DW_TAG_variable
                    DW_AT_name  ("_r")
                    DW_AT_type  (0xe024 "__regs *")
                    DW_AT_location      (DW_OP_fbreg -24)
```

Listing 6.6: The DWARF debugging information does not contain any information about the values stored in registers during the execution of the function.

However, this would require knowledge about the data flow within the assembly code. The tool would have to figure out the surrounding type by following the data-flow in reverse until it finds a register or memory location with an associated DIE. Then it would have to follow the data-flow back to the original memory access operation. As part of this it would be necessary to map the semantics of low-level operations back to the high-level semantics on the C type level. For instance, an indirect memory load would represent a pointer dereference. Alternatively, more precise debug information from the compiler could also solve this problem. These would have to include information about every intermediate register value in the program.

The same applies for the other samples we investigated closer, and the problem is responsible for the most of the failures to recognize types of heap allocations. Not being able to recognize the type of some heap allocations increases the risk of false-positive constant detections. We observed this happening in the file system code where all vnodes were falsely detected as being of the same type. Injecting the constant caused the unikernel to being unable to read any files from the file system.

We previously discussed the importance of having a high coverage during the tracing phase. During our investigation of the problems of some constants in the nginx benchmark, discovered that the combination of all individually working constants does not work. Debugging the unikernel revealed that nginx attempted to write text to the standard error as shown in Listing **??**. The concrete file descriptor is abstracted away using the `ngx_open_file_s` type, which stores the file descriptor in the `fd` field. While tracing only a single value was observed for this field. Therefore, it was determined to be a constant as seen in Listing 6.8. This value corresponds to an `epoll` file descriptor during run-time.

```
n = ngx_write_fd(log->file->fd, errstr, p - errstr);
```

Listing 6.7: Nginx uses a common structure to store information about open files such as log files and event notification file descriptors.

```
- type: type
  target: ngx_open_file_s
  path: ["fd"]
  value: [3, 0, 0, 0]
```

Listing 6.8: Our analysis tool only saw a single value for the file descriptor part of this open-file structure.

The result is that the `write` system call is called on this file descriptor. This erroneous usage causes a crash of the unikernel. While a more extensive test setup during the tracing phase could prevent this kind of error, it is hard to assess when the test setup is sufficient to cover all cases. Instead, a more reasonable approach would be to specifically white-list known configuration structures. This case shows that there are structures where the fields only change their values at very specific circumstances. This is in contrast to the normal configuration structures where the values stay the same across the runtime of a program.

6.4 Conclusion

Overall, the detection of constants in global variables worked well except for the memory layout description variable. We attribute this to the precise debug information about global variables that allows us to accurately track values in them. The binary size reduction by injecting the detected constants was rather small compared to the total size of the unikernel.

The constants that were detected on a type-level were more problematic but also had more potential for space savings. We think that a more robust detection of types in heap allocation would also translate in a more robust detection of constants in them. This in turn should also improve the detection of constants on a type-level. The current implementation for the type detection of heap allocations is not enough for real-world applications as a considerable effort is necessary to verify the binaries.

Chapter 7

Conclusion

Unikernels have shown a considerable benefit over traditional appliances based on full classical operating systems by allowing extensive specialization. Instead of having to support a broad set of possible applications, they can omit features unnecessary for the targeted workload. This allows them to reduce the overhead from an image size, run-time memory usage, and attack surface point-of-view.

A common pattern is to port an existing application such as a web server to unikernels. These applications are often built to satisfy many usage scenarios and often use configuration files to allow the user to specify the desired behavior. The program usually parses the configuration file at startup and then uses the resulting values during runtime. In unikernels this configuration is often bundled within the final image and never changes during deployment. Even modern compilers with link-time optimizations cannot utilize that these values are effectively constant and remove unnecessary code for disabled features.

Our approach is to use a memory trace to find out the final (constant) values in the structure used to store the parsed configuration. To accomplish this we use the debugging information emitted by compiler to reconstruct the structures used by the program during runtime. In the reconstructed structures we can observe changes to the values by working through the memory trace and note what fields in structure stay constant across the execution.

The resulting list of constants can be then passed to a modified compiler which replaces any read to those with the constant itself. This allows the compiler to effectively do constant propagation from configuration files into the program code. We showed that this approach can fundamentally work and has a positive impact on the binary size.

7.1 Future Work

Our current approach is not able to replace function pointers, because the concrete pointer values changes when the function moves around in the binary. This already happens when the size of a function shrinks because of our optimization and all following functions move to lower addresses. Instead, the optimization would have to instruct the linker to insert the correct value. A possible implementation would be to introduce special globally visible alias-symbols that allow referring to internal-linkage symbols. Because some build systems aggressively hide all non-whitelisted symbols, this would also require an integration into those affected build systems.

A limitation of our current analysis tool is that it currently cannot do any data-flow analysis to figure out more types of heap-allocations as discussed in Section 6.3.1. This limitation is also responsible for some incorrectly detected constants and limits the usefulness of our tool in practice.

Bibliography

[1] Tim Bird. *Advanced Size Optimization of the Linux Kernel*. Apr. 2013. URL: https://elinux.org/images/9/9e/Bird-Kernel-Size-Optimization-LCJ-2013.pdf (visited on 01/03/2022).

[2] Hadi Brais. *Compiler Optimizations - Streamline Code with Native Profile-Guided Optimization*. Sept. 2015. URL: https://docs.microsoft.com/en-us/archive/msdn-magazine/2015/september/compiler-optimizations-streamline-code-with-native-profile-guided-optimization (visited on 01/07/2022).

[3] M. Burtscher et al. "The VPC trace-compression algorithms." In: *IEEE Transactions on Computers* 54.11 (2005), pp. 1329–1344. DOI: 10.1109/TC.2005.186.

[4] D. R. Chakrabarti and Shin-Ming Liu. "Inline analysis: beyond selection heuristics." In: *International Symposium on Code Generation and Optimization (CGO'06)*. 2006, 12 pp.–232. DOI: 10.1109/CGO.2006.17.

[5] Pohua P. Chang, Scott A. Mahlke, and Wen-mei W. Hwu. "Using Profile Information to Assist Classic Code Optimizations." In: *Softw. Pract. Exper.* 21.12 (Dec. 1991), pp. 1301–1321. ISSN: 0038-0644. DOI: 10.1002/spe.4380211204.

[6] Zakir Durumeric et al. "The Matter of Heartbleed." In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. IMC '14. Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 475–488. ISBN: 9781450332132. DOI: 10.1145/2663716.2663755. URL: https://doi.org/10.1145/2663716.2663755.

[7] DWARF Standards Committee. *The DWARF Debugging Standard*. URL: https://dwarfstd.org/ (visited on 06/26/2022).

[8] Free Software Foundation, Inc. *GNU Compiler Collection (GCC) Internals*. URL: https://gcc.gnu.org/onlinedocs/gcc-3.4.6/gccint/index.html (visited on 06/28/2022).

[9] Galois Inc. *The Haskell Lightweight Virtual Machine (HaLVM): GHC running on Xen*. URL: https://github.com/GaloisInc/HaLVM (visited on 06/18/2022).

[10] *includeos/IncludeOS: A minimal, resource efficient unikernel for cloud services*. URL: https://github.com/includeos/IncludeOS (visited on 06/18/2022).

[11] Pavel Kosov and Sergey Yakushkin. *LLVM PGO Instrumentation: Example of CallSite-Aware Profiling*. Sept. 2020. URL: https://llvm.org/devmtg/2020-09/slides/PGO_Instrumentation.pdf (visited on 01/07/2022).

[12] Simon Kuenzer et al. "Unikraft: Fast, Specialized Unikernels the Easy Way." In: *Proceedings of the Sixteenth European Conference on Computer Systems*. EuroSys '21. Online Event, United Kingdom: Association for Computing Machinery, 2021, pp. 376–394. ISBN: 9781450383349. DOI: 10.1145/3447786.3456248.

[13] Hsuan-Chi Kuo et al. "A Linux in Unikernel Clothing." In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387526. URL: https://doi.org/10.1145/3342195.3387526.

[14] Anil Madhavapeddy et al. "Unikernels: Library Operating Systems for the Cloud." In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. Houston, Texas, USA: Association for Computing Machinery, 2013, pp. 461–472. ISBN: 9781450318709. DOI: 10.1145/2451116.2451167. URL: https://doi.org/10.1145/2451116.2451167.

[15] Filipe Manco et al. "My VM is Lighter (and Safer) than Your Container." In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pp. 218–233. ISBN: 9781450350853. DOI: 10.1145/3132747.3132763. URL: https://doi.org/10.1145/3132747.3132763.

[16] Joao Martins et al. "Enabling Fast, Dynamic Network Processing with ClickOS." In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. HotSDN '13. Hong Kong, China: Association for Computing Machinery, 2013, pp. 67–72. ISBN: 9781450321785. DOI: 10.1145/2491185.2491195. URL: https://doi.org/10.1145/2491185.2491195.

[17] Microsoft Corporation. *Microsoft/microsoft-pdb: Information from Microsoft about the PDB format*. URL: `https://github.com/Microsoft/microsoft-pdb` (visited on 06/26/2022).

[18] Pierre Olivier et al. "A Binary-Compatible Unikernel." In: *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE 2019. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 59–73. ISBN: 9781450360203. DOI: `10.1145/3313808.3313817`. URL: `https://doi.org/10.1145/3313808.3313817`.

[19] Donald E. Porter et al. "Rethinking the Library OS from the Top Down." In: *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, Inc., Mar. 2011. URL: `https://www.microsoft.com/en-us/research/publication/rethinking-the-library-os-from-the-top-down/`.

[20] Marc Rittinghaus, Thorsten Groeninger, and Frank Bellosa. *Simutrace: A Toolkit for Full System Memory Tracing*. White Paper. Karlsruhe Institute of Technology (KIT), Operating Systems Group, May 2015.

[21] Colin Robertson, Kent Sharkey, and David M. Gillies. *Profile-guided optimizations*. Microsoft. Aug. 3, 2021. URL: `https://docs.microsoft.com/en-us/cpp/build/profile-guided-optimizations?view=msvc-170` (visited on 01/07/2022).

[22] *rumprun/rumprun: The Rumprun unikernel and toolchain for various platforms*. URL: `https://github.com/rumpkernel/rumprun` (visited on 06/24/2022).

[23] *runtimejs/runtime: Lightweight JavaScript library operating system for the cloud*. URL: `https://github.com/runtimejs/runtime` (visited on 06/18/2022).

[24] Lorenzo Stanco. *lorenzos/Minixed: A minimal but nice-looking PHP directory indexer.* URL: `https://github.com/lorenzos/Minixed` (visited on 07/14/2022).

[25] Frank Tip and Jens Palsberg. "Scalable Propagation-Based Call Graph Construction Algorithms." In: *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '00. Minneapolis, Minnesota, USA: Association for Computing Machinery, 2000, pp. 281–293. ISBN: 158113200X. DOI: `10.1145/353171.353190`.

[26] Chia-Che Tsai et al. “Cooperation and Security Isolation of Library OSes for Multi-Process Applications.” In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys ’14. Amsterdam, The Netherlands: Association for Computing Machinery, 2014. ISBN: 9781450327046. DOI: `10.1145/2592798.2592812`. URL: `https://doi.org/10.1145/2592798.2592812`.

[27] Liang Wang et al. “Peeking Behind the Curtains of Serverless Platforms.” In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 133–146. ISBN: ISBN 978-1-939133-01-4. URL: `https://www.usenix.org/conference/atc18/presentation/wang-liang`.