Diplomarbeit

zur Erlangung des akademischen Grades
Diplom-Informatiker

Compiling Unikernels into Micro Kernels

Lisza Zeidler

Advisor:	**Dr.-Ing. Sebastian Ertel**
1. Examiner:	**Prof. Dr.-Ing. Jeronimo Castrillon** Chair for Compiler Construction, TU Dresden
2. Examiner:	**Dr.-Ing. Sebastian Ertel** Composable Operating Systems Group, Barkhausen Institut

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tag dem Prüfungsausschuss der Fakultät Informatik eingereichte Diplomarbeit zum Thema:

Compiling Unikernels into Micro Kernels

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Dresden, den 31. Januar 2023

Lisza Zeidler

Abstract

With increasing complexity, programs become more prone to bugs and security vulnerabilities. This is particularly true of kernels. For example, the original feature set of the monolithic Unix kernel is still continuously extended by functions, drivers and modules. Since these are not mutually constrained, each additional component increases the attack surface of the entire system. There are several approaches to solving this problem and implementing the concept of defense in depth. However, they all involve runtime costs and, most importantly, manual customization. This effort makes it difficult or impossible to flexibly adapt existing software to isolation mechanisms that provide an appropriate trade-off between security and performance overhead.

The idea of this work is to investigate whether the isolation of components of a server application can also be done by a compiler. The Ohua compiler has been developed to identify independent execution steps in a sequential program and to transform the program into a dataflow program consisting of independent nodes with potentially separate memory. The specific implementation of the nodes is determined by architectural integrations. Nodes can be threads or processes, or theoretically other isolation concepts. We wanted to use Ohua to convert a server application, in which the application, IP stack and network interface share the same memory, into an application for the microkernel-based operating system M^3 . The main questions were i) how to restructure the server application so that in the resulting dataflow graph the application, IP stack and network interface each operate in exactly one isolated node, and ii) could these restructurings be implemented as compiler transformations.

We show how the application can be restructured accordingly. Formal descriptions already exist for some of the transformations. However, it has also become clear that the syntax of the input program alone does not contain enough information to define, for example, whether or not the program should be adapted to concrete target systems such as M^3 . Therefore, this thesis discusses which transformations could be implemented as compiler transformations in the future, and which transformations still have to be done by the programmer.

Contents

1 Introduction

Whenever tasks become more extensive and complex, systems react with division of work and specialization. By subdividing into smaller, less complex tasks, large processes can be distributed and processed in parallel, and at the same time the executing components are more efficient because they have to cover less noise, peripheral cases and side aspects. We can observe this development on different scales in computer science. At the hardware level, computations are distributed from general-purpose CPUs to GPUs, digital signal processors (DSPs) or other adapted hardware, and the concrete hardware logic is abstracted from the program logic to be executed. At the software level, components are separated both vertically, i.e. between business logic, language runtime environment and operating system services, and horizontally, i.e. between components of the different levels of independence.

Besides improved efficiency and scalability, compartmentalization has also a major advantage when it comes to security. Separation of concerns and minimization of trust assumptions are core concepts in defense in depth for cloud deployments. But again they are likewise seen on the scale of single machines. A cloud deployment consisting of one big trust zone protected by a perimeter is the large scale equivalent of a monolithic kernel, with unrestricted memory access among all kernel space processes. The problem with this is clearly evident in monolithic Unix kernels. Despite various measures to prevent unauthorized memory access (e.g. control flow integrity and data execution prevention) or to make it more difficult to exploit security vulnerabilities (e.g. address space layout randomization and stack canaries), according to statistics of the NIST[1] the number of vulnerabilities in the kernel ecosystem is still increasing. An analysis of the NIST National Vulnerability Database in [38] looked at the reasons for critical vulnerabilities over the last 5 years and concluded that 34% of them were due to lack of storage security, and another 43% were due to lack of compartimentalization. The authors also pointed out that these vulnerabilities could have been prevented if essentially independent processes could not access shared memory, but that however the so-called least-privilege policy is not enforced by the Linux kernel or other big open source projects as OpenSSL or the Apache Server. Similarly, as the authors report in [27], the majority of known vulnerabilities in Windows, Chrome and the Android Open Source Project (AOSP) can also be traced back to insecure memory access.

So obviously it is desirable to have compartmentalization and data locality enforcement not only in cloud setting but also in the kernel itself. Existing solutions for memory protection are tied to specific hardware requirements as Memory Protection Keys, Trusted Execution Environments like ARMs Trust Zone[40] or Intels SGX [12] or specific virtualization layers ([45], [39]) and in general require manual code adaptations. The costs of adapting tends to make migration between architectures and to newer solutions more difficult and leads to components being subdivided more coarsely than would make sense in order to save effort and runtime costs. Also in terms of security, compartmentalization and in particular concurrency comes at a cost. Recent trends in secure computing are strongly moving away from simple

testing towards verification and proving. This applies to the proof of certain properties of user programs as well as to the verification of compilers, kernels, or operating systems ([33],[28], [20]). Distributed, concurrent programs, however, are much more difficult to verify.

While certain costs are unavoidable when program parts are isolated, the costs of manually adapting to different isolation mechanisms are not among them. The classic approach to separating concrete runtime implementations from program logic is compilers. They allow the programmer to define and test the logic within specific programming models and automatically translate and add to them to fit the chosen runtime environment and architecture. This work will therefore also be based on Ohua[17], a compiler developed to identify independent processing steps from sequential programs and deploy them to concurrent, potentially isolated nodes of a data flow graph. Ohua works on subsets of high-level languages such as Python or Rust. Instead of machine code, Ohua extracts a dataflow program from the input, introducing two main abstractions, the notion of an independent node and the notion of communication edges, and replaces them with the corresponding implementations for different run times. The second aspect of Ohua is the ongoing effort to formally describe and verify the transformations it applies. The goal is to be able to verify sequential input programs and transform them into a distributed program using Ohua, without losing the guarantees of verification.

The idea of this work is to use and extend Ohuas capabilities for program transformation. We want to be able to extract isolated components from a shared memory program suitable for unikernels and automatically derive the code to deploy them in a microkernel setting. Concretely we will use the M^3 operating system as an example backend to provide process isolation. The key questions we need to answer are:

> How can we rewrite a program written for monolithic or unikernels into one in which isolated components work together in a data flow graph?
>
> Can we generalize these refactorings to compiler transformations?

A good example case to approach the answer to this question is are server applications. They are quasi-ubiquitous in distributed applications and consist of various components, in particular the data backend, the TCP/IP-Stack and a network driver that should be isolated from each other for security reasons. Using this example we make the following contributions in this work:

- We present a simple server application based on smoltcp and discuss basic structural problems that stand in the way of compiling into independent components.
- We describe how these structural problems, such as visibility of object usage, shared memory access or stack management when splitting methods, can be solved for the concrete example.
- We discuss which of the conversion steps are generalizable, how they could be formalized, and the resulting requirements for the programmer.
- We propose approaches to extend the supported syntax of Ohua and the functionality of the M^3 backend.

To better understand the starting point and requirements of this work, in Chapter 2 we first consider the function of Ohua and the properties of the target architecture M^3. Certain properties of Rust's type system are also explained in this chapter, as they form the basis of Rust's storage security and are helpful constraints for meaningful transformations. On this basis, Section 3 describes the our example application. In a rough sketch, we approach the structural problems and describe schematically how the application should function after the transformations. We then describe how individual aspects of the code must be transformed. After looking at some related approaches in Chapter 4, we discuss what we can learn from the applied refactorings. Specifically, in Chapter 5 we list the main problems targeted, to discuss whether and how they could be solved by compiler transformations. We also address possible implementations for language features that Ohua does not currently support. Chapter 6 briefly concludes our findings.

2 Background

The goal of this work is to understand which transformations are necessary to convert a program from a sequential programming model with shared memory into a distributed, concurrent programming model of a microkernel operating system. A significant part of these transformations is already implemented in the Ohua compiler. Therefore, the specific goal is to understand and close the 'translation gaps' between the programming models of the source program, the current Ohua implementation, and the M^3 operating system. So to better understand the work that needs to be done, we will introduce those models in this chapter.

2.1 Ohua

In this section we will introduce the Ohua compiler[17]. Its essential concept is to extract the underlying data flow graph from a sequential input program. The result is a program structure consisting of individual steps that are connected only by incoming and outgoing data and can be executed concurrently. These individual steps and their data channels can then be mapped to various abstractions of processes and communication channels in the backend of the compiler to achieve parallelization and isolation of the steps. Like most compilers, Ohua works step-by-step with different intermediate representations (IRs) of the input program. To be able to support different languages and target architectures, Ohua uses language integrations. Currently there are integrations for Rust and Python. In the next subsection, we will take a closer look at the basic structure and function of Ohua.

For this work, it is also important to understand what programming model Ohua currently supports. Here, programming model means on the one hand which restrictions in the syntax of the input programs are currently necessary to be able to convert them into correct concurrent programs. On the other hand it contains the explicit and implicit assumptions about the concrete implementations of 'processes' and 'channels'. So we will also look at these aspects in more detail in this section.

2.1.1 Compiler Pipeline

When we say 'Ohua compiles a program', we mean it compiles functions in the compile scope. In contrast to e.g. rustc or gcc, Ohua does not compile the complete code of the program, but transforms only the functions, for example within one or more Python modules specified in the call. We call these functions algorithms. In contrast to algorithms, functions and methods that are imported and used within algorithms are not compiled. They are completely opaque to the compiler. This also means that Ohua does not require any syntax constraint in these imported functions and methods. An overview of the compiler pipeline is shown in Figure 2.1.

In the **compiler frontend**, algorithms of the target language are first parsed and translated into Ohua's frontend language (Frontend IR). This translation is implemented in language specific frontend integrations. That is, for each language supported by Ohua, such a frontend integration must exist. This integration parses algorithms of the input language and translates them into the syntax of the frontend IR. For non-supported syntax constructs, currently for example `break` statements in loops, the compilation terminates at this point. The currently supported subset of Rust can be found in the Appendix A.1.

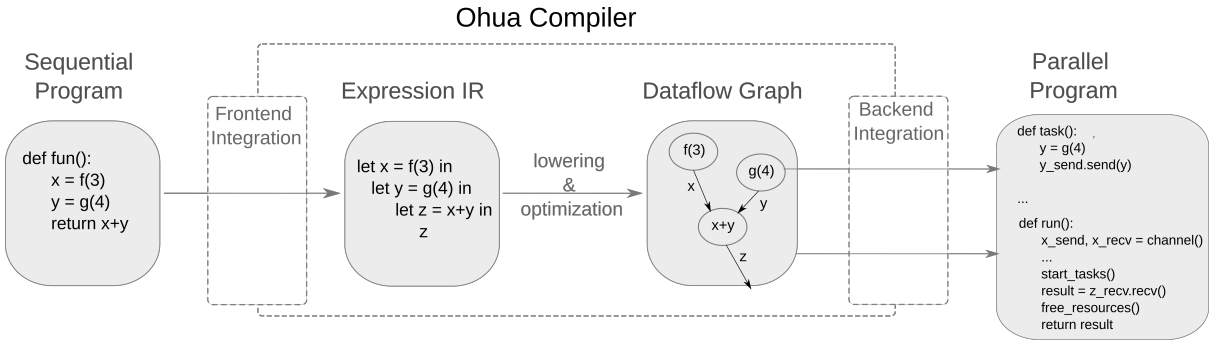


Figure 2.1: Structural overview of Ohua and the code transformations

In the **core compiler** itself there are two main representations of the code. One is Expression IR. This language is functional and based on the call-by-need lambda calculus. To transform input algorithms to this representation calls to other algorithms are inlined, renaming compiler passes ensure single static assignment form, and assignment expressions are refactored to applicative normal form. A simplified example of code before and after the transformation is shown in Listing 2.1. A central conversion step in the compiler is the transformation of stateful calls to so called *state threads*([47], [31],[16]). To generate race condition free tasks, Ohua forbids shared state use and only permits stateful computation inside methods. However, methods in an imperative language mutate objects in place and implicitly refer to the new, changed state by the same reference as before. The conversion of stateful calls to state threads makes the semantic of creating a new state upon calling methods explicit. In the code example in Fig.2.1, the call `someState.do()`, is internally transformed to explicitly take a state as an argument and return a new, mutated state. Thereby, function calls downstream to not need to access a shared memory to use stateful objects. Based on this explicit state threading, Ertel et al. [16, 15] developed functional representations for imperative control flow on stateful computations. For example, an imperative for-loop is translated to a so called `smap` operation, which is essentially a fold operation of the loop body on the states manipulated inside the loop.

The next representation in the compile flow is the Data Flow Graph (DFG) representation. Independent program tasks are encapsulated in the this representation and are explicitly assigned their incoming outgoing data channels. Besides function calls from the original program, this representation also contains control nodes that govern the data flow. For example if the input code contained a branching statement like `if cond {f()} else {g()}` control nodes will be introduced to a) switch data flow between calls to `f()` or `g()` and b) to collect results from appropriate output channels of `f()` or `g()` depending on the condition. This representation also allows to merge certain nodes by fusing their code, as well as input and output channels. This is done for instance if a following node entirely depends on its predecessor and has so little work to do that it would hardly justify the overhead of spinning up an independent

```
fn algo(i){
  let someState =
          other_algo(i);
  let a = someState.do()
  let b = f(a)
  return b
}

fn other_algo(i){
   let s = State::new(i)
   s
}
```

(a) Input algorithm

```
let someState =
        λ State::new (i) in
  let a, someState_0 =
           λ do (someState, a) in
    let b = λ f (a) in
        b
```

(b) Pseudocode of IR

Listing 2.1: An algorithm is mapped to a nested let-expression with the innermost term representing its return value

task in any backend implementation.

Which ultimately brings us to the **compiler backend** and backend integrations of Ohua. Backend integrations consist of two parts. The 'Language Backend' is only language specific. Similar to the frontend integration, it serves the purpose of translating code inside tasks from Ohua representation syntax back to the target language syntax. The 'Architecture Backend' is responsible for translating the 'nodes' and 'edges' of the DFG into a specific implementation for concurrent tasks, communication channels and a runtime for the graph. For example, there can be two different architectures for a given language: one implementing tasks as threads and one using processes, with both also generating appropriate channels and runtime code to execute the DFG. As we did not compile imported functions, the target language must match the input language, which is automatically ensured by the compiler. Architectures for the same language can be used interchangeably. This way Ohua can generate e.g. multi-threaded shared memory or fully distributed programs from the same input.

In the next section we will take a closer look at the restrictions and assumptions required to ensure that compilation works as expected.

2.1.2 Programming Model

The term programming model generally describes a relationship between syntax constructs in a programming language and their concrete semantics in a particular execution environment. In the case of Ohua, the programming model includes, on the one hand, the supported input syntax and the assumptions made about the supported terms of the input language. On the other hand, it specifies how these terms are translated into a dataflow graph and what assumptions Ohua makes about the implementation of nodes, edges, and runtime of the DFG. First we will look into the supported input syntax and assumed semantics.

2.1.2.1 Input Syntax and Semantics

We already know that Ohua's basic input units are algorithms, i.e. pure functions inside the compile scope. Table 2.1 depicts the language definition of the frontend representation described before. Any syntax construct of the input language has to be mapped to the according terms of this language to be compiled. In the following paragraphs we will describe the accepted syntax constructs and the semantics the programming model expects them to have.

Patterns:			
p	::=	$x \mid (x, \ldots, x) \mid ()$	named variables, tuples or unit
Expressions:			
e	::=	e	named expression in host language
	|	**1**, **2**, **3**, ... | **true** | **false** | **()**	typed literal in host language
	|	**let** p **=** e **in** e	lexical scoping
	|	$e\ e$	application
	|	$\lambda[p, \ldots, p]$**.** e	abstraction
	|	**if** e **then** e **else** e	conditionals
	|	**map** $e\ e$	map first expression to second
	|	**bind** $e\ e$	bind an expression representing a state to an expression representing a function to act on this state
	|	**stmt** $e\ e$	expression whose return value is ignored
	|	**seq** $e\ e$	
	|	**(** e **)**	tuple of expressions

Table 2.1: Definition of the Expression IR

Function Calls: Beside algorithms, Ohua supports stateful and stateless function calls, i.e. methods and pure functions, imported into the scope. Pure functions are expected to be side effect free. In particular, the programming model assumes that pure functions do not implicitly manipulate their arguments. This excludes, for instance, functions that manipulate their arguments by reference. If the output of a pure function call is not used, it is considered to have no effect and is removed during compilation.
Stateful function calls, on the other hand, are expected to manipulate the object they are called on, i.e. have a side effect. Consequently, they are not removed regardless if the output is used. Any stateful computation is expected to happen exclusively and explicitly using method calls and also method calls are expected to only manipulate the object state itself. This also entails the requirement that the state is not 'leaked' via return values. For example, in the method call `let x = SomeState.do_stuff();`, `x` must not be, or contain, a reference to `SomeState`. We already assumed that other functions do not manipulate `SomeState` when using `x` as argument. However, without this 'leaking assumption' it would be possible to call `x` as a stateful object, thereby implicitly manipulating the state of `SomeState`. This implicit semantic is not handled currently and would be lost in the distributed output code.
Functions need to be typed or type-able by the frontend integration. To correctly annotate the types in generated code, at least to the extent required for any particular backend and architecture, Ohua needs to extract type information from the input code. Specifically the argument types of each function call are extracted and preserved in the different IRs as typed function literals capturing the argument types of each function call.

Loops: Ohua supports bound and unbound `for`-loops. They are transformed into parallelizable pipelining of the independent calculation steps inside the loop. Inside for-loops, each state from outside the loop must be used at most once to enable the accumulation of state changes in a single node for each object. Conditional loops (`while` or `do-while`) are as of the beginning of this work not supported, but can be expressed using recursion.

Recursion: Ohua supports recursion with some notable restrictions. Recursive algorithms must be tail recursive, the recursive call must be located in the if-branch, and the return value must be in the else-branch of recursion. Furthermore, either one must be the only statement in each branch respectively. As return values only single variables are supported. Recursive loops will not yield any pipelining or parallelization but a loop executed for one input at a time. This is due to the semantic of recursion, being a repeated function on a state, where the result of each step depends not only on that step but also on previous results. Currently, the output of recursive algorithms cannot be used in an assignment, i.e. a recursive call can only be the last statement in another algorithm and return the calling algorithm's final result.

Branching: Branching is supported in case of simple if-else expressions, where both branches must be present. Also, if-else statements, as for instance present in the Python syntax, are currently not supported. This is because those statements have a different execution semantic than expressions, i.e. branches that do not return a value but have side effects on variables from the surrounding scope, so they need to be implemented separately. In addition, it is currently not possible to use stateful functions in branches.

Return, Continue, Break: Ohua currently does not support any forms of early return of conditional execution except for recursion and branching as described before. Therefore, `break` and `continue` are generally not supported at the moment, while `return` is supported only for Python and only at as the last statement of a function block, because contrary to Rust there is no implicit value return in Python.

Variables and Literals: There are two categories of variables. Local variables are bound inside algorithms, environment variables are bound in outer scope. Thus, environment variables are basically arguments of the algorithms, but can also be imported or globally defined names. Ohua supports mutable and immutable local variable bindings. Local variables can either be used as a state, or as an argument to a function call. If it is used as an argument it can only be used once, if it is used as a state, it may be used more than once except, as explained before, inside loops. Environment variables cannot be used as a state directly[1], but can be used several times as function call argument. The underlying assumption of this distinction is that environment variables will be available in scope for all nodes created from an algorithm, while locally bound variables are sent to the consuming node. This assumption becomes relevant in architectures where the generated tasks have no access to a common global scope. In those cases, environment variables are not available to the task via a closure mechanism. This will be the case for M^3 tasks. Finally there is a limited set of literals that is directly supported in the input. This includes integers, booleans, strings and unit literals. Other literals must be wrapped in a function call

[1]This changed during the course of this work. Now arguments to algorithms can be used directly as a generator for a for-loop.

currently (simple binary operations are sufficient here e.g. to compile `let x = 17` one could write `let x = 17 + 0`).

Many of the current limitations are only due to lack of implementation. For example, there is no formal reason to allow recursive calls only in the if branch. These limitations will be fixed in the future. At the moment, however, they are the main reason why control flow expressions cannot be freely combined. That is, there are currently restrictions on the frontend language that are not reflected in the language itself.

2.1.2.2 Enforcement

Conformity with the allowed syntax subset is automatically implemented by each language integration, as it has to translate the input syntax to the frontend language. However, this is only a syntactical conversion. Except for tracking the annotated type of named variables, the compiler does no further semantic analysis of the input code. This means that to comply with a programming model requirement it is sufficient to match the expected syntax, but not necessarily the expected semantics. Take for example the requirement to use each variable only once as a function input. To use a variable `x` twice, a programmer needs to return it twice from a function call `let (x1, x2) = fun()`. However, she can freely decide whether `x1` and `x2` are copies or only references of `x`, matching the expected syntax but not the expected semantics of the programming model. The result might still be valid output, but it is the responsibility of the programmer to ensure validity of reference passing in the concurrent output. Likewise, the use of global mutable state can be encapsulated in function calls.

In general, enforcement of the programming model is currently not separately implemented and in some case lacking completely. Violations of the programming model will either lead to runtime errors during any point of compilation or may lead to invalid output code. The latter case was not an obvious problem in Rust, as Rust itself enforces borrowing rules and therefore a considerable part of Ohua's limitations. However, this is not generally the case in other languages, so tests were added in the course of this work to ensure compilation failure upon violations of the programming model.

2.1.2.3 Backend Language and Process Abstractions

The terms of Ohua's backend language are depicted in Table 2.2. As complex syntax constructs of the input language are removed in the frontend, most of the terms in the backend language are basic constructs of any imperative language, as variables, literals, assignments, and simple control flow terms. To generate the code for Ohua-introduced control nodes, some *specific functions* are also required to be present in the host language. In particular, the control of loop execution requires implementations for list handling, as well as the functionality for *iterable objects* of the host language, to test if they have a fixed size and if so retrieve this size information.

Obviously, the backend language also entails the notion of named and typed channels, their sending and receiving ends, and sending and receiving of variables a means of communication among the tasks of the data flow graph.

Typed Task Expressions:			
e	::=	x |**1**, ... | **true** | ()	variables, simple literals
	|	**funRef** $\|ref$**envRef** $!HostExpr$	function and environment references
	|	$x(e,\ \ldots,\ e)\|\ Obj.x(e,\ \ldots,\ e)$	pure function and method calls
	|	**let**x **=** e **in** $e\|$ **=** e	scoped bindings and assignments
	|	**stmt** $e\ e$	statement
	|	x**_receiver.receive()**	expression to receive data
	|	x**_sender.send(**x**)**	expression to send data
	–control flow–		
	|	**while True:** e	
	|	**for** x **in** x **do** e	
	|	**repeat** $(x \mid l)\ e$	
	|	**while** $e\ e$	
	|	**if** e **then** e **else** e	
	– specific functions, required for control nodes –		
	|	**newList**	create a list
	|	**append** $x\ e$	append t to x
	|	**hasSize** x	$[a] \rightarrow$ Bool
	|	**size** x	$[a] \rightarrow$ Int
	|	**(** $l,\ l$**)** | **(** $x,\ x$**)**	Tuple of literals or bindings
	|	$(x,\ \_)$	First
	|	$(\_,\ x)$	Second
	|	$x++$	Increment
	|	$x--$	Decrement
	|	**not** e	
Communication Channels:			
channel	::	**channel** x	Typed channel, i.e. incoming and outgoing end for variable x
	|	x**_receiver**	Typed receiving end of a channel
	|	x**_sender**	Typed sending end of a channel

Table 2.2: The terms of Ohuas backend language used to represent the DFG. Language specific backend integrations translate this language to generate the output program.

A major advantage of compiling with Ohua is that the generated dataflow-based language is deterministic. That is, a correct, deterministic, sequential program becomes a correct, deterministic, concurrent program by compilation. Formal verification of the compiler transformations to proof this claim and formalize the programming model is currently an ongoing task. Nevertheless, we can already clearly describe the assumptions concerning the concrete implementations of nodes, edges and the runtime each architecture must provide.

Specifically, we assume for the implementation of nodes and runtime that:

1. Nodes do not share mutable memory. However, the architecture provides access to environment references, which may be global constants, imports, and (most importantly) the arguments of the compiled algorithm.

2. There can be more nodes than the runtime is capable of running concurrently and there is no explicit scheduling. Therefore we assume cooperative multitasking, i.e. nodes waiting for input will free computation resources for other nodes.

3. The runtime instantiating the nodes is capable of ending them and freeing resources.

Since it is a data flow language, the execution of the programs is controlled by the data flow. This results

in the following assumptions for the implementation of the edges:

i) all data are transferred in order,

ii) there is no implicit use of default arguments (default arguments are in general possible, but there has to be an explicit signal for every computation in a node and every parameter it uses that this parameter is 'None' and should be replaced by the default value for this particular execution round/loop),

iii) receiving is blocking (this closely relates to ii) in the sense that there must not be a calculation or result passed on while it is not clear whether a term of that calculation just has not arrived at the moment of calculation),

iv) all types visible in the compile scope are sendable in the given architecture.

Contrary to the assumptions on the input code, the assumptions about the architecture and backend integration cannot be validated inside the compiler.

2.2 Micro- and Unikernels

General-purpose operating system have to provide a broad range of functionalities to connect application layer to hardware layer. This includes user interfaces (graphics), networking, security, device drivers, and most obviously the functionality required for program execution and memory management, which may also include virtualization mechanisms for programming languages as Java and Python. To enable efficient execution and communication between the components, Unix-like operating systems, for example, are often implemented as monolithic kernels. In monolithic kernels the system services (daemons) and drivers have direct access to the hardware and the shared memory.

However, this approach has considerable downsides. Since drivers and daemons run in kernel mode, they have full rights and access to the resources of all other processes. This means that in the event of a vulnerability in one of the components, the entire system is affected in principle. In particular third-party device drivers were notoriously faulty and a portal for exploits[2]. Also, many applications do not require most of the services provided by those general purpose systems. For example, a microservice that merely answers simple requests to a key-value store only needs the functionality of the network stack and the file system. Libraries and system functions for additional user or device interfaces only unnecessarily increase the complexity, memory consumption and attack surface of the service.

Two alternative concepts of kernels are micro- and unikernels. Figure 2.2 shows how user applications, drivers and system services, and the hardware layer are compartmentalized in each of these kernel architectures in principle. Basically, the idea of microkernels is to reduce the code run in kernel mode to the absolute minimum required to access the actual hardware layer, while unikernels are often based on a microkernel, but also give non-essential components required for a specific app to run direct access to

[2]Recent example of driver bugs in the Linux kernel

hardware resources. We will briefly introduce the two concepts, as well as the related concept of library operating systems here.

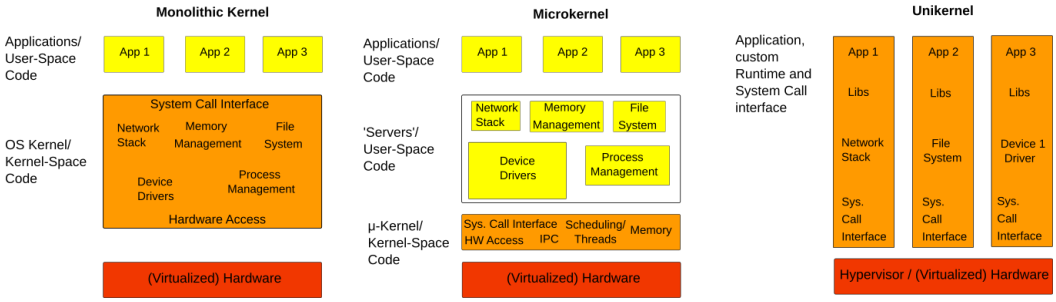


Figure 2.2: Structural comparison of Monolithic, Micro- and Unikernels

Microkernels► Microkernels follow a *minimality principle* formulated by Liedtke [26] as

"More precisely, a concept is tolerated inside the µ-kernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality."

The central motivation for microkernels is the reduction of privileged, low-level code executing in kernel mode. Based on the insight that large code bases of monolithic kernels come at the cost of a large potential for bugs in privileged, low-level (and therefore hard to check) code, concepts to minimize kernels (and therefore attack surface) date back to the 70^th [21]. Concepts like the Mach kernel [6], separation kernels[42], or isolation kernels [48] where developed to minimize and isolate kernel space code, to increase security, and enable verification.

The minimality principle basically limits the essential components of kernel-to-memory management (i.e. providing access and access control to address spaces), CPU allocation (i.e. providing access to the CPU in any form of process or thread abstraction and scheduling), and inter-process communication (IPC). Other services, such as I/O, device drivers, networking, and others are run as userland processes, although there might be further distinctions from actual user processes. In addition to the advantage of the smaller attack surface, the low memory requirement also makes microkernels advantageous, especially for embedded systems.

This design comes with an inherent performance penalty. In a monolithic system a userspace application requiring access to hardware or system services would cause a single context switch to kernel mode. The request would be answered and the result returned to the userspace process. In a microkernel, however, the request of the user application will be forwarded by the kernel via IPC, e.g. to a driver process that again answers via IPC indirected through the kernel. In this simple scenario, the number of context switches doubles from two to four. Also, the communication among system services is just function calls in monoliths, while it again involves IPC and four context switches for each invocation among services.

Currently existing examples of microkernels are L4 (formerly L3 [34]), Minix [22], Singularity [24] or the QNX microkernel OS[23] used in embedded systems for example in phones, or as real time OSs in

cars.

Unikernels[36] ►: Unikernels also tackle the problem of large code base and attack surface in monolithic kernels. However, they follow a different approach concerning process isolation. The 'uni' in unikernels refers to the idea of compiling a specific kernel for each application or even component of larger applications. Basis for compilation is a library operating system written in rather high level languages, the user program, and a configuration file to specify the target architecture and the required library components. Library operating systems provide functionalities of monolithic kernels as independent library implementations. For example, device drivers for physical NICs are implemented in libraries that can be combined with potentially different implementations of the TCP/IP stack. Examples of library operating systems are MirageOS[35], Graphene [46], IncludeOS [10] or Unikraft [29]. In the compiled unikernel, all processes run with kernel privileges and have direct access to the hardware or hardware abstraction layer (e.g. a hypervisor). This reduces size and attack surface compared to monolithic kernels and has, unlike monolithic and microkernels, no IPC overhead for context switches. It is not well suited and not intended to be used for multi-user scenarios. However, large applications can be realized with unikernels by distributing the app components into several distinct unikernels. In cloud applications, this setup allows the hypervisor to scale only required parts of the application. Examples of this principle are CubicleOS [44] and FlexOS [32].

2.3 The M^3 Operating System

2.3.1 The Concept

M^3[9] is a microkernel concept/architecture for distributed and potentially heterogeneous architecture, as for example different embedded processors cooperating in modern cars. It comprises a hardware and a corresponding software, i.e. OS and kernel design. Specifically, the hardware design describes the components necessary to connect and control separate chips, e.g. for broadband communication, signal processing (camera, GPS), or cryptographic operations, while the corresponding operating system provides communication and access to the hardware for the apps running on each component.

An overview of the design of M^3 is shown in Figure 2.3. The architecture is composed of tiles, communicating with each other via data transfer units (DTU). There is only one tile that runs the actual microkernel. This tile is also the only one requiring a general purpose core (GPC) as underlying hardware. The computing units (CU) inside the other tiles might be general purpose CPUs, FPGAs, DSPs or fixed-function accelerators. Processes on CUs run independently and isolated from each other. They can however communicate to each other via the DTUs. The kernel is responsible for scheduling tasks on the CUs. Running tasks are called activities. On tiles using CPUs, an activity is basically a running system thread. Via the DTUs, the kernel can also control context switches between activities on the CUs and establish communication relations between activities. By default, activities run in their own address space and are disconnected from each other. To establish a connection, an activity A would once request the kernel to connect, for instance to an activity B running the network stack. Once that connection is established, the activities can directly communicate without involving the kernel again, which eliminates some of the communication overhead in other microkernel systems.

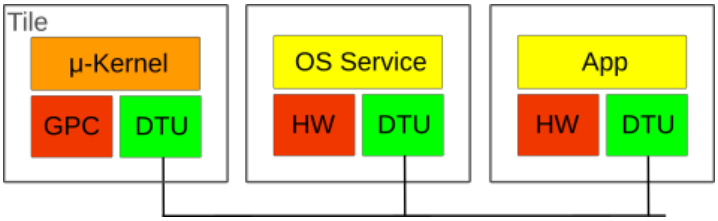


Figure 2.3: The m3 architecture is composed of tiles, communicating via data transfer units (DTUs).

In Section 2.1.2.3 we discussed the assumptions Ohua makes about the backend architectures' tasks and channels. In particular, we noticed that tasks have to be cooperatively scheduled and channels have to ensure in-order, guaranteed delivery. Here we take a look if these requirements are fulfilled by M^3.

When activities are idle, they notify the kernel, which can then schedule another activity. So cooperative scheduling is given. Communication channels are unidirectional, first-in-first-out connections. Activities are not aware if their communication partner is currently running or suspended i.e. if the context was switched by the kernel. The hardware (i.e. the DTU of the receiving tile) detects attempts to communicate with a not-running activity and errors back to the sending activity. Upon such error, the sending activity invokes the kernel to schedule the required receiving activity. The kernel buffers messages for the receiver in this case. Despite this fallback mechanism, the authors state that message delivery has only-once semantic, so data access can/must be repeated if necessary upon failure.

2.3.2 The Rust API

M^3 provides a Rust integration to access its abstractions of processes and channels. Here we briefly describe the main features of this API. A simple example of how an activity can be instantiated is shown in Listing 2.2[3]. Activities can be initiated either on a common tile or on several tiles. In the example, the tile of the parent process is used. The API uses a closure syntax to define activities. However, those definitions have no closure semantic, i.e. they do not enclose definitions from the surrounding scope.

To pass capabilities and file descriptors to a child activity, the attribute `activity.data_sink()` can be used to serialize data into the process memory. This data can be accessed from inside the activity code using `data_source()`. Serialization is implemented in a custom `Serializer` based on the `serde` crate. We need this mechanism, as shown in the example, to pass send and receive gates to the activities. For simple environmental variables, the API provides a separate mechanism using `m3::env`. This basically provides a global key-value store in which variables can be stored before the definition of an activity, retrieved by key inside the activity, and deleted after the activity definition to not pollute the namespace.

Concerning process communication, the API provides different mechanisms (mainly depending on the size of the data to be transmitted) of which direct sending over channels is the most relevant to us. M^3 manages communication among processes using capabilities. The capability to directly send to or receive from another process is implemented as `Gates` in the M^3 Rust API. Gates are synchronous, di-

[3]Example adapted from M^3 Rust unit tests

```rust
fn run_send_receive(t: &mut dyn WvTester) {
    // Get a descriptor of the current tile
    let tile = Tile::get("clone|own");
    // Initialize a new activity on the current tile
    let mut activity = ChildActivity::new_with(tile, ActivityArgs::new("test"));

    // initialize send and receive gate with message order, size and credits
    let rgate = RecvGate::new(math::next_log2(256), math::next_log2(256));
    let sgate = SendGate::new_with(SGateArgs::new(&rgate).credits(1));

    // make receive gate avaible in the activities namespace
    activity.delegate_obj(rgate.sel());
    let mut dst = activity.data_sink();
    dst.push(rgate.sel());

    // define activity
    let activity = activity.run(|| {
        ...
        // make receive gate available inside activity
        let mut src = Activity::own().data_source();
        let rg_sel: Selector = src.pop().unwrap();
        let rgate = RecvGate::new_bind(rg_sel);

        // receive
        let mut res = recv_msg(&rgate));
        let i1 = res.pop::<u32>();
        let i2 = res.pop::<u32>();
    });

    send_vmsg!(&sgate, RecvGate::def(), 42, 23));
}
```

Listing 2.2: Example of creating and running a Rust activity on M^3

rected one-to-one connections, so there are send gates `SGate` and receive gates `RGate`. Receive gates are instantiated with a maximum message size and a maximum overall size of the message buffer to prevent memory overflows in limited environments. The absolute (system immanent) maximum message size is 2 kb. Send gates are instantiated with a number of credits. With each message send, one credit is used. The receiver can return those credits by answering on a received message. Note that this mechanism is hidden inside the `recv_msg` and `send_vmsg!` calls. Both are or contain macro calls to generate and receive responses upon message receipt to pass back sending credits.

Apart from the credit system, sending messages is straight forward, as known from most pipe-like interfaces. Receiving is done in two steps. First a receive stream is requested using `let stream = rgate.recv_msg()`. This is a blocking call, that will return upon available messages arriving at the gate. The second step is calling `let msg = stream.pop()`, which is non blocking. This call can be done arbitrarily often on an existing stream but will fail if there are no more messages in the stream. Finally, a limitation of the current M^3 API is that the standard library is not supported. There is a separate implementation of `std` with essential functions but partly different function signatures. For smoltcp itself this limitation is not relevant, since it is written without using the standard library. In contrast, the libc API is a necessary part of smoltcp and is supported by M^3 .

2.3.3 Existing M^3 Architecture Integration

Ohua already features an architecture integration for Rust on M^3 . It is built on a simplified API of M^3 that provides two main encapsulations, the `channel()` call and the `activity!` macro. Listing 2.3 shows how an activity can be created using those functionalities. The function `channel` basically wraps the initialization of pairs of send and receive gates with default message orderings, a default message size, and a default of one send credit. This allows to create channels basically the same way as in pure Rust or Python. The `activity!` generates code to 1) instantiate a `ChildActivity` on the current tile, 2) delegate the given gates to the activity, and 3) bind and activate the gates inside the activity. This again basically resembles the initialization of a thread in pure Rust.

```
use funs::hello_world;

fn test() -> String {
   use m3::com::channel::{Sender, Receiver};
   use m3::activity;
   let (a_0_0_tx, mut a_0_0_rx) = channel();
    activity!(
        (|a_0_0_child_tx: Sender| {
            let a_0_0 = hello_world();
            a_0_0_child_tx.send(a_0_0)?;
            Ok(())
          }
        )(a_0_0_tx)
    );
    a_0_0_rx.activate()?;
    a_0_0_rx.recv::<String>().expect("Error message")
}
```

Listing 2.3: Example of creating an activity using the simplified M^3 Rust API

However, as previously explained, activities in M^3 are not closures. This means that in order to use environment variables, an additional mechanism will be necessary. This is currently not part of the architecture integration.

2.4 smolTCP

smolTCP [5] is a Rust-based open-source library for network stack implementations. It runs entirely as a user space application. It also provides conditional compilation features to build applications without heap allocation. This makes smoltcp and applications built with it amenable to be used in microkernels such as M^3[8] and embedded systems such as ARTIQ (e.g. [30]). In fact, the microkernel operating system Redox[2], as well as M^3 itself, already use smoltcp for their network stack implementation. So in both systems, smoltcp is used to create a network stack as a sequential, userspace service. This means that, in contrast to the application to be developed in this thesis – the TCP/IP layer, the actual network layer or system interface and the user application communicate with each other via shared memory.

The design of the library is structured according to typical TCP/IP layering. We will briefly introduce the three main layers or components, respectively, that are relevant in this work[4].

[4]For more information see the documentation at `https://docs.rs/smoltcp/latest/smoltcp/`.

The `socket` module provides different socket abstractions implementing the TCP, UDP, IGMP or DHCP protocols, respectively, as well as for raw sockets. Common features of all those abstractions are keeping track of inbound and outbound data in socket buffers and implementing functionality to package or unpack those data according to the implemented protocol. The sockets keep also track of additional state information, relevant for their respective protocol e.g. TCP or DHCP client states, hop limits, window sizes, etc.

The `iface` module provides the abstractions of the IP layer. The most important structure is `Interface`. In an inner component of the interface (`InterfaceInner`), the state data of the IP layer are stored. This includes the IP address of the interface, a routing table, a neighbor cache, and the hardware address (depending on the transport medium according to Ethernet or IEEE 802.15.4 standard). Accordingly, the generation and interpretation of IP headers for outgoing and incoming packets are also tasks of the `InterfaceInner`. In the currently official variant of smoltcp, the Interface also contains a field `Device`, holding an abstraction of the physical network layer, and a `SocketSet` to manage all sockets belonging to the interface. Smoltcp is under ongoing development and so, in the recent implementation state, `Device` and `Sockets` are no longer part of the interface, but independent structures passed to it to process packets from sockets to device and vice-versa. We will discuss the implication of this in further detail in the Section 3.1.

Finally, the physical layer is implemented in the `phy` module. It provides different implementations of the `Device` trait, to connect the application to the underlying operating systems loopback or tuntap interface or raw sockets. Implementors of the `Device` trait provide the methods `receive`, `transmit` and `capabilities`.

The actual transfer of packets from the interface to the device is realized via sending and receiving tokens. We explain this in a little more detail here, because it exemplifies a characteristic of the smoltcp code. A successful call to `device.receive()` will yield a tuple of a receive token `RxToken` and a send token `TxToken`. The former will contain the actual received content as a private field, the latter contains a reference to the device specific storage for outgoing packets. In case of tuntap and raw-socket devices, this reference is a file pointer provided by the operating system. In Lising 2.4 the a slightly simplified implementation of a sending `TxToken` for raw sockets and the usage of such token to send an Ethernet frame are shown. Two things become apparent. First, the memory for the packets is allocated only at device level upon consuming a token. Second, any structs needed to construct the frame are instantiated in a closure passed as an argument to `tx_token.consume()`. As closures are realized via fixed size structs on the stack frame of the called function, this technique enables smoltcp to work without heap allocation and is used heavily in the code. This principle is used in a cascading manner i.e. `dispatch_ethernet` itself also receives a closure capturing objects from the calling scope.

Obviously, one implication of separating the layers of smoltcp to different, memory separated components is that this kind of memory efficiency cannot be maintained.

```rust
// Sending Token for RawSocket
pub struct TxToken {
    lower: Rc<RefCell<sys::RawSocketDesc>>,
}

impl phy::TxToken for TxToken {
    fn consume<R, F>(
    self,
    ...,
    f: F) -> Result<R>
    where
        F: FnOnce(&mut [u8]) -> Result<R>,
    {
        let mut lower = self.lower.borrow_mut();
        let mut buffer = vec![0; len];
        let result = f(&mut buffer);
        match lower.send(&buffer[..]) {
            Ok(_) => result,
            // Error handling
        }
    }
}
```

```rust
//Interface using a TxToken to send a frame
pub fn dispatch_ethernet<Tx, F>(
&mut self,
tx_token: Tx,
buffer_len: usize,
f: F) -> Result<()>
where
    Tx: TxToken,
    F: FnOnce(EthernetFrame<&mut [u8]>),
{
    let tx_len = EthernetFrame::buffer_len(buffer_len);
    tx_token.consume(self.now, tx_len, |tx_buffer| {
        ...
        let mut frame = EthernetFrame::new(tx_buffer);
        let src_addr = {...};

        // closure from outer scope:
        f(frame);
        Ok(())
    })
}
```

Listing 2.4: Sending and receiving packages is implemented via Tokens exposing a `consume` method, taking closures as argument that process the send or received content. This way, memory allocation can be constrained a) to the device layer and b) to the stack if necessary.

2.5 Rust

The main questions of this thesis are i) What requirements must a sequential program meet in order to be converted into a concurrent program in a semantics-preserving way? and ii) What steps are necessary for the conversion? Rust as a programming language is particularly well suited to investigate these questions. This is due to Rust's extended type system that cannot only enforce well-typedness of a program but also statically ensure the validity of references and therefor safety of memory usage at runtime. The two central concepts enabling this are *ownership* and *lifetimes*. *Ownership* is used to ensure memory safety in (safe) Rust. In particular, it allows the Rust compiler to ensure at compile time that no two pieces of code can modify an object at any time, i.e. that at runtime there is never more than one reference to an object that allows write access. *Lifetimes* extend the concept of ownership to track the validity of references. This model does not only prevent frequent errors in sequential programs, it also enforces already a substantial part of the assumptions of a distributed programming model. Therefore, necessary restrictions of a distributed, concurrent programming model are particularly well to be examined in Rust, because they are already enforced in the sequential code. In Chapter 3, we will refer to those concepts and their implications for our findings. So to better understand these implications, we will first take a closer look at the rules of ownership and lifetimes here.

Ownership: Rust has neither garbage collection nor does the user need to free and allocate memory herself. Instead, Rust's runtime manages memory using ownership and lifetime rules. The Rust compiler ensures, for most types statically, that there is exactly one *owner* of each value, i.e. one variable holding it, created either on heap or stack. When this owner goes out of scope (i.e. when the scope it has been declared in ends), the value is cleared from memory without the need for extensive reference tracking garbage collection. This is realized by automatic implementations of the `drop` function for every type. A call to `drop` is added automatically by the compiler for each variable as it goes out of scope.

Ownership has a different implementation for values located on the stack and on the heap. For values stored on the stack, the owner is simply the variable assigned to that value. For a value allocated on the heap, the variable owning that value holds a pointer to the memory location of that value and additional

information as for instance actual and totally allocated size of the value. This difference is important for the semantics of passing values as function arguments or returning them from functions. The act of passing values directly, i.e. not only passing a reference, is called *moving* in Rust parlance. For both the stack based and the heap based values, *moving* means copying the owner, i.e. copying the information the owner holds on the stack, to the stack frame of the called function. Returning values also follows that principle. But, as we noticed before, for stack based values this actually also copies the value itself. This results in a new value, with a new owner, and allows the old owner to remain valid. For heap based values on the other hand, only the memory reference to the value is copied that way. So, to prevent having several owners addressing the same memory location, the old owner gets invalidated upon *moving*. To actually duplicate heap data, one needs to implement or derive the `Clone` trait for the data type. An explicit call to the `clone()` method will then deepcopy the heap allocated data and its owner, resembling the semantics of implicit copy on stack based values.

In a distributed scenario, passing a value to a function call will involve transferring that value to another process's memory that we do not expect to sync with the original location. So, as it is enforced by the Rust compiler, we cannot use the original reference to that value any more, since it would lead to inconsistent states. In fact, Ohua's programming model, at least theoretically, is more strict than that of Rust. While it always assumes pure function arguments to be used immutably, i.e. read-only, it also currently requires variables to be used only once.

References: Obviously, besides passing the value directly, Rust also offers the possibility of creating references to both stack and heap based values. In contrast to pointers, Rust's references are guaranteed to address valid data. References do not convey ownership. In Rust's memory model this means, references are just an address to the owner, and do not contain information about the size and capacity of the data pointed to. Passing values by reference is called *borrowing* in Rust terminology. References, just like values, are immutable by default. The general rule for borrowing a value is that at each point in the code there can only be either one mutable or an arbitrary number of immutable references to each owner. References are values themselves, i.e. they have an owner and a lifetime tied to the scope they are valid in. So Rust can track validity of reference owners to enforce the borrowing rules. Beyond scope validity, the rust compiler is capable of identifying the last use of a reference and shorten its lifetime to this point (a concept called 'non-lexical lifetimes'). Therefore it is possible to borrow a value mutably multiple times within a scope. When one mutable reference is created and used only after the last use of another one, the compiler can shorten the former one's lifetime to end before the next reference becomes valid.

Lifetimes[5]: As explained before, references do not convey ownership. So *lifetimes* are Rust's way to ensure that the owner a reference is pointing to is not dropped while the reference is still in use. In general, every reference has a lifetime. Inside a single function scope, lifetimes are mostly implicit. The compiler can simply derive them from the local scoping. However, when references are used as an argument or return value, lifetimes cannot in general be derived by the compiler and need to be made explicit. Note that the constructors of `enum` and `struct` types are essentially also just functions, taking

[5]We will focus on the main aspects for this work here, but a full introduction to Lifetimes, as well as rules for subtyping and inference can be found in the Rust Book[3] and the Rust Reference[4] in the Chapter *Subtyping and Variance*.

fields as arguments and returning the respective `emun` or **`struct`**. So obviously they also have to abide lifetime rules.

As we noticed, lifetimes are meant to ensure reference validity. The mechanism is best explained using the case of function return values. Returning valid values, either by value or by reference from a function is only possible if the owner of that value will not be dropped at the end of the function. This is the case if

a) The function returns a moved owner: The returned value is allocated on the heap and the actual owner is returned. This will result in a move, copying the owners reference information to the return stack frame.

b) The functions returns a copied owner: The returned value is allocated on the stack, but implements `Copy`. This will copy the value itself to the return stack frame.

c) The function returns a reference to the owner: The actual owner of the value lives in the outer scope and is manipulated by reference in the function. If a reference to the original owner is returned, the owner will continue to be mutably borrowed for the lifetime of this reference. This also includes references to static memory, e.g. string literals known at compile time.

When are lifetime annotations needed? The simple answer is: whenever there are references and the compiler cannot infer them automatically. The first two cases generally do not require lifetime annotations. In case a) the data live on the heap, but the 'owning information' is copied to the return scope so Rustc can automatically infer its lifetime further on by tracking the owner. In case b) the data is copied to the stack of the return scope and again the compiler can automatically infer scopes and lifetimes. In case c), however, the lifetime of the reference returned is bound to the lifetime of the memory location (reference) passed as a parameter. The reason why not every function accepting and/or returning references needs lifetime annotations in parameters, return value, and the function itself is the so-called 'lifetime elision'. Rust uses a set of rules to automatically annotate lifetimes in standard situations, which are

1. In a function declaration all references are assigned a freshly generated lifetime parameter
   ```
   fn fun<'a, 'b, ..., 'z>(a:&'a A, b:&'b B, ...) -> &'z Z
   ```
2. If there is only one input reference, its lifetime is also assigned to the output reference.
   ```
   fn fun<'a>(a:&'a A) -> &'a Z
   ```
3. If the function is a method all outputs are assigned the lifetime of the object acted on.
   ```
   fn fun<'self, 'a, ... >(& 'self self, a:&'a A,...)
    -> (&'self Z1, &'self Z2, ..)
   ```

The same lifetime rules that hold for function arguments and returns basically hold for structs. A **`struct`** holding a reference in a field requires that reference to come from a surrounding scope. Just as a function cannot take a reference that might not live as long as the function runs, a **`struct`** can only live as long as the shortest living reference it holds. The same holds for **`enum`**s, obviously. Just like for functions, this means that those **`struct`**s and **`enum`**s are generic over at least one lifetime parameter, determined by the reference they are given on creation.

When those rules are insufficient, annotations are needed. For example if there are multiple input references and a specific one should provide the lifetime information for the output. Also automatic inference might lead to unwanted effects. For instance, if a reference to a `static` value is passed to a function, the compiler will infer the returned reference to also live for `static`. This means the original value cannot be borrowed mutably or even immutable again in the program. In this case lifetime annotations help the compiler to solve lifetime unification at all or in a more favorable way.

So when and why would we need to consider lifetimes? We expect the programmer to provide valid input Code to the compilation. Also, the programming model of Ohua prohibits reference arguments in scope. So up to this point, there was no necessity to consider lifetime implications in the transformations Ohua applies to the code. However, in this work we will not only manipulate the code 'in scope' but also inside the components that will later be opaque to Ohua. It is possible, and for efficiency reasons desirable, to keep on working with references inside those components. Meanwhile, we will also introduce and in particular split existing functions. Therefore, we will need to consider constraints and effects of reference lifetimes on our transformations.

3 Implementation

So our declared goal is to transform an application, written for a shared memory, single core runtime into a form that is amenable to compilation with Ohua. Also the resulting compiled program should have the following properties:

1. The application should have three independent, stateful components, namely a key-value store answering requests, the TCP/IP stack and the device driver.
2. Those components can run concurrently, i.e. it is possible to streamline process packages.
3. Those components are local to a single executing process each, i.e. no statefully used component is ever send from one process to another.
4. The final data flow program runs on the M3 operating system and architecture.

To achieve this goal, both smoltcp and Ohua must be adjusted. Therefore, in this chapter we will first describe the structure of our smoltcp application and the necessary changes. Then we will describe how the current functionality of Ohua needs to be adapted and extended.

3.1 Transformations in smoltcp

In its current sequential, single-threaded state, smoltcp makes use of several language features that Ohua's programming model does not support because they cannot be translated into a deterministic concurrent program. The first problem is the sharing of memory references between different components. If these components are executed concurrently, race conditions occur. Roughly speaking, it is therefore necessary to replace all accesses to shared references with explicit data flow.

The second fundamental problem is the current flow of control between stateful components. In a sequential program we can invoke a method on one component A, do something on it's state, internally call a method on another component B and finish the outer call by maybe changing As state again. A and B belong to the memory of the same process and the state of A as well as the execution state of the outer method are automatically preserved during the call of B. This is what happens when a packet is sent or received using smoltcp in its current form. However, in a concurrent, distributed data flow graph, we cannot realize this bi-directional dependency among stateful components. Why is that? The main problem here is realizing concurrency and state locality at once. We could connect distributed components A and B using blocking connections, i.e. A calls B and blocks until it receives the result. What we achieve this way is a distributed, sequential program. Another way would be to have A send its state along with the call to B and B would return this state along with its answer. So A can process the next input and upon receiving an answer it can resume the appropriate state. However, this contradicts the requirement of state locality. In particular when states are large objects e.g. a database state, a training state of a deep

neural network or alike, we want them to stay encapsulated in on component.

So to compile to a distributed program with local components of smoltcp, we need to disentangle state dependencies from each other. In the following subsection, we explain how this can be done using a small example key-value store application. We begin with introducing the concrete program and sketching the structure of the target program.

A simple server application: In Listing 3.1 we see the code for our simple key-value store application. First the required structures, i.e. the `ip_stack`, the `device`, the `store` and some sockets are initialized. Then in the `ip_stack.poll` call the socket and the device are used to send the messages from the sockets to the device and vice versa. Then, depending on the state of the socket, the store answers received messages directly to the socket. Finally the application holds for a time interval determined by a) the current state of the sockets `ip_stack.poll_delay` and b) the current state of the device, represented by waiting for its file pointer `fd` to become available. After that, the interface exchanges messages again.

```rust
fn main() {
    let mut store = Store::default();
    let mut device = TunTapInterface::new("tap0", Medium::Ethernet).unwrap();
    // ... more interface initialization code
    let mut ip_stack = builder.finalize(&mut device);

    let mut sockets = SocketSet::new(vec![]);
    // .. more socket initialization code
    let tcp_socket = tcp::Socket::new(tcp_rx_buffer, tcp_tx_buffer);
    let tcp_handle = sockets.add(tcp_socket);

    loop {
        let timestamp = Instant::now();
        match ip_stack.poll(timestamp, &mut device, & mut sockets) {
            Ok(_) => {}
            Err(e) => {
                debug!("poll error: {}", e);
            }
        }

        let socket = sockets.get_mut::<tcp::Socket>(tcp_handle);
        if !socket.is_open() {
            socket.listen(6969).unwrap();
        }

        if socket.may_recv() {
            let input = socket.recv(process_octets).unwrap();
            if socket.can_send() && !input.is_empty() {
                debug!(
                    "tcp:6969 send data: {:?}",
                    str::from_utf8(input.as_ref()).unwrap_or("(invalid utf8)")
                );
                let outbytes = store.handle_message(&input);
                socket.send_slice(&outbytes[..]).unwrap();
            }
        } else if socket.may_send() {
            debug!("tcp:6969 close");
            socket.close();
        }

        phy_wait(fd, ip_stack.poll_delay(timestamp, &sockets)).expect("wait error");
    }
}
```

Listing 3.1: Simplified example of a key-value server application using smoltcp

Obviously, this code does not meet our compilation requirements. Concrete problems are

1. There are several stateful uses of `socket`, e.g. in `socket.listen(6969)`. However, we do not want those to be visible to the compiler, since they would result in their own, stateful nodes. This is basically a problem of efficient resource utilization, as every node incurs the overhead of an

independent process or thread of the target architecture.

2. In the call `ip_stack.poll` the `device` is passed as an argument. This means the compiler cannot *see* the device being used as an independent component and will not derive the according code.

3. The `ip_stack` is statefully called twice inside the loop. This means the compiler will not be able to derive a single node acting on/as the `ip_stack` but instead wil derive a data flow where the `ip_stack` is send from one node using it to the other.

4. We cannot simply transfer the mechanism of `phy_wait(fd,...)` to another operating system. For one thing, the function itself is a system call, which means it depends on the operating system. Secondly, `fd` is a file pointer and the handling of pointers and files is also strongly dependent on the target operating system and architecture.

So given this initial code setting and the problems we need to solve to enable the compilation, we can now proceed with describing the actual changes made in this work.

3.1.1 Sketching the Target Structure

To better understand the transformations described in this chapter, we will figure out the structure of the target program first. We want each of the target components to be used exactly once inside the scope as an object a method is called on. Further we want all implicit state sharing, i.e. the common use of references between components to be eliminated and be replaced by actual explicit message passing. Finally we want function calls, other than the three stateful calls and the initialization of our object to happen outside the compile scope in order to restrict the resulting data flow graph to as few nodes as possible and to generate an efficient program.

The first step to get a clearer picture of our goal is to wrap code details the compiler does not need to see into function calls. So as a first step and to get a clearer picture of our target structure, we encapsulate the initialization of components. Also we saw in the original code in Figure 3.1 `sockets` being processed, *loaded* with messages by the `store` and than passed to the `ip_stack` along with the `device` for actually processing the TCP/IP packets. So one could figure to use the `sockets` are the vehicle we use for communication among the `ip_stack` and the `store` for now and encapsulate the socket handling in the `store.handle_message` call. Also we let the store application handle the result of `ip_stack.poll`. Now the code looks as in Listing 3.2:
Now to get an idea of the target structure without going to much into detail, we make the following assumptions:

i) We assume for the time being that there is a simple control flow inside `ip_stack.poll` going from the `ip_stack` to the `device` for sending, back to the `ip_stack`, and

ii) We know that the `phy_wait` call takes an input from the device, namely the file pointer `fp` and an input from the `ip_stack`, namely a waiting time from the sockets and halts the loop for a certain interval before the `ip_stack` can poll again. Given this structure, we assume we can prepend the calls involved in `phy_wait` to the execution of `poll` itself. Applying this assumptions, we get a control flow

```
fn main() {
    let (mut store_app, mut sockets):(App, SocketSet) = init_app_and_sockets();
    let (mut ip_stack, mut device, fd):(Interface<'static>, TunTapInterface, RawFd) =
      init_stack_and_device();

    loop {
        let timestamp = Instant::now();

        let (poll_result, sockets_polled) =
          ip_stack.poll(timestamp, &mut device, sockets);

        let sockets_loaded = store_app.handle_sockets(poll_result, sockets_polled);

        sockets =
          phy_wait(
            fd,
            ip_stack.poll_delay(timestamp, sockets_loaded))
            .expect("wait error");
    }
}
```

Listing 3.2: Server application after encapsulating code into objects

as depicted in Figure 3.1. We see an inner loop among the `ip_stack` and the `device` and an outer loop among the `ip_stack` and the `store`. As smoltcp can be used heapless, all process steps act on shared references instead of moved owned values.

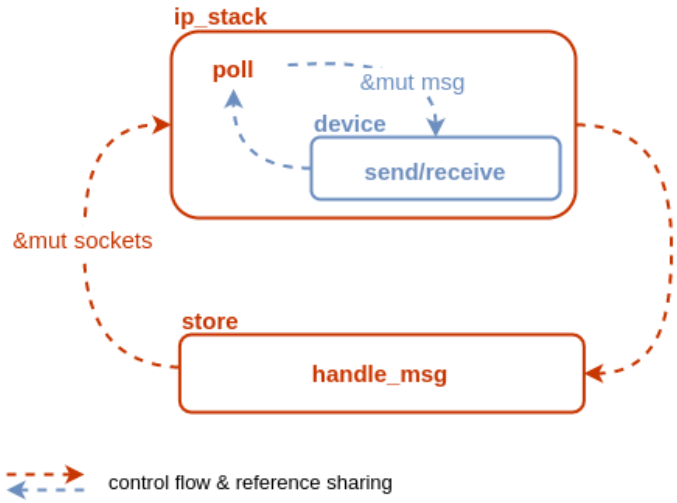


Figure 3.1: Simplified structure of the original process loop

Now for this simplified structure, we can already see what needs to be done to achieve our target characteristics. We need to move the inner loop inside `poll` into the main scope. For now we assume we had simply split `poll` into parts e.g. asking the `device` for availability, waiting before poll, preparing packets and sending them. And we create a method `ip_stack.process()` that can execute either of those parts, based on the input. Also we change all method signatures to explicitly return and except values instead of references.This directly results in the structure shown in Figure3.2.

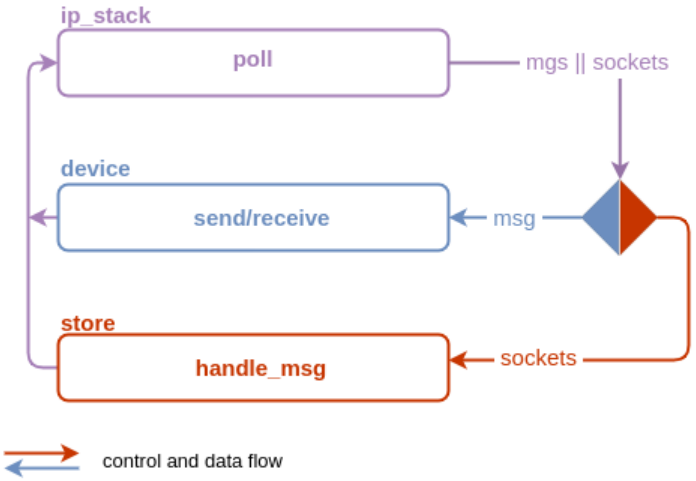


Figure 3.2: Simplified structure of the target process loop

Without looking at the inner workings of poll in detail, we can already recognize essential properties of the target program here

1. The final structure is a nested loop.
2. Since we want to use only one method call for the `ip_stack`, it must return a type which can be either an input for the `device` or the `store`.
3. The calls to `device` or the `store` must return a common type that `ip_stack.poll` accepts[1].
4. In the original structure, the `device` was called inside the `poll` function. Therefore, after each call to `device`, the control flow automatically returned to the execution state of `poll`. Now the substeps of `process` are individual method calls. That means we will need a way to transfer the execution state of `poll` from one call to the next.

In the next sections, we will first take a closer look at the control flow in poll. We will describe how to implement inner loop extraction and what problems arise when converting reference arguments to moved objects, i.e. call by reference to call by value.

3.1.2 Encapsulating the Socket Handling

In sketching the target program, the first change made to the original code was to encapsulate as much of the logic as possible into either the states or the initialization functions. As a first guess, the socket handling i.e. checking if sockets are open, setting them to listening state etc. was encapsulated in the `store.handle_sockets` call. The problem with this decision is that the actual `SocketSet` type contains lifetime bound references, meaning they do not implement `Clone` and we cannot just replace passing them by reference with actually moving them.

[1] you could also return different types. However, then an additional node would be needed to convert them to a common type

Implementing a custom, owned version of the `SocketSet` would be a remedy for this problem. However, the internal references are actually different buffer types used in the sockets and manipulated in many places in the core logic of smoltcp. This means that implementing a custom, owned `SocketSet` type would not only result in extensive changes, but would also negatively affect the efficiency of the program. The other option is to not send the sockets, but only the required information among components. What this required information is, depends on how each of the components involved interacts with the sockets. For our example key-value store application, the interaction between sockets and `store` is fairly simple. The `store` receives a message from a socket and answers it. As we will later see, the interaction between the `ip_stack` and the sockets is rather involved. So instead of having a free-floating `SocketSet` in the main scope, we make it part of the `ip_stack` upon initialization. This also means, we need to move the code for handling the sockets into the `ip_stack` as well and return to the `store` not directly after polling, but when a socket is processed. We will discuss how this is done in Section 3.1.4. The main point here is to illustrate the problem of designing the interfaces between components, if they are not clearly defined in the first place.

A less simple case could have involved, for instance, conditional opening or closing of sockets by the `store` based on concrete messages, time, traffic volume etc. and would have required an extended message interface between the `ip_stack` and `store` and potentially a replication of the sockets in both components updating each other via this interface. In general, this information cannot be derived from static, purely syntactic information from high-level language input code.

3.1.3 Refactoring Packet Sending

The next step is to refactor the `poll` method. We need to bring the stateful usage of the `device` into the compile scope and create a single *entry method* the three components use to call each other. This *entry method* will internally dispatch each call to the actual method, or generally code to be called on the states.

The code of `poll` is depicted in Listing 3.3. The method itself contains a loop calling `socket_egress` which executes sending packets to the `device` and `socket_ingress` which receives packets and distributes them to the `sockets`. As long as there are packets send or received in each loop, `poll` continues.

As we can see the `device` reference is passed further down into two method calls, the receiving method `socket_ingress` and the sending method `socket_egress`. We will start with explaining the transformations in `socket_egress`, because it is structurally more complex than `socket_ingress` and therefore better suited to demonstrate all necessary transformation steps.

3.1.3.1 Lifting `device` calls into Scope

Listing 3.4 shows how sending packets proceeds as a loop over the socket items. If a `socket` can send, its `socket.dispatch` method is called passing a reference to the `ip_stack` and the `respond` closure. As our example works with TCP sockets, inside `dispatch` the `socket` prepares a TCP packet and passes it to the `respond` closure. This closure will then try to get a sending token from the `device` and pass the token and the TCP packet. The token is basically a direct reference to the sending buffer of the

```rust
pub fn poll<D>(
    &mut self, timestamp: Instant,
    device: &mut D, sockets: &mut SocketSet<'_>,
    ) -> Result<bool>
    where
        D: for<'d> Device<'d>,
    {
        self.inner.now = timestamp;
        let mut readiness_may_have_changed = false;

        loop {
            let processed_any = self.socket_ingress(device, sockets);
            let emitted_any = self.socket_egress(device, sockets);
            if processed_any || emitted_any {
                readiness_may_have_changed = true;
            } else {
                break;
            }
        }
        Ok(readiness_may_have_changed)
    }
}
```

Listing 3.3: Simplified code of the `ip_stack.poll` method

`device`. Finally inside the `inner.dispatch` call of the `stack` will wrap the TCP packet into a network representation e.g. an Ethernet frame and directly write that packet into the device buffer using the token.

The `device` is used twice in this method. Once for receiving a token in the `device.transmit()` call, a second time implicitly when this token is used to pass a packet to the device inside the `inner.dispatch_ip` call. The both of this calls occur inside the `socket.dispatch` method, when the `respond` closure is called. To resolve the nested structure we start with inlining the code of the `respond` closure to `socket.dispatch` and get the structure shown in Listing 3.5.
Dispatching a packet starts with a preparation phase were the socket state is checked. If no packet can be produced, it returns with `Ok(())`. If a packet is produced, a token is required and if successfully created the packet is sent. Afterwards the socket state is updated using information from the packet. If any of the steps fail, the control flow returns an error to the outer scope i.e. to the sending loop.

To lift device usage into compile scope, we need to inline the content of `socket.dispatch`. This means we refactor a function scope i.e. a separate stack frame to a simple scope in the outer stack frame. Scopes are expressions in Rust. So there are two different ways of returning values from scopes. One is to simply end the scope with an expression. In this case, the scope simply returns the value of that last expression to the outer scope. The second is to use explicit `return` or `?`. They will not only return a value or an error to the surrounding scope, but leave the current stack frame.

So to inline function with early returns or `?`, we need to refactor the early returns to conditional execution i.e. we need to

1. instantiate a `let result;` at the beginning of the method and whenever either a `return x` or a *expression*`?` appears

```rust
fn socket_egress<D>(&mut self, device: &mut D, sockets: &mut SocketSet<'_>)
-> bool
{
    let Self{inner, ..} = self;
    let mut emitted_any = false;
    for item in sockets.items_mut() {
        if ! can_send(item, inner.now) {
            continue;
        }
        let mut neighbor_addr = None;
        let mut respond = |inner: &mut InterfaceInner, response: IpPacket| {
            neighbor_addr = Some(response.ip_repr().dst_addr());
            let token = device.transmit().ok_or_else(|| {
                Error::Exhausted
            })?;
            inner.dispatch_ip(token, response, None)?;
            emitted_any = true;
            Ok(())
            };
        let result = match &mut item.socket {
            Socket::Tcp(socket) => socket.dispatch(inner, |inner, response| {
                respond(inner, IpPacket::Tcp(response))
            }),
            // ..handle other socket types
        };
        match result {
            // stop looping if the device was exhausted
            // update socket if adress was wrong
        }
    }
    emitted_any
}
```

Listing 3.4: Simplified code of the `ip_stack.socket_egress` method

2. replace any returning statement including the final return by binding the potential early return values to the `result` variable, in our example for instance
   `result = inner.dispatch_ip(token, response, None);`
3. wrap every subsequent code into a conditional and,
4. return the assigned `result`

After this refactoring, the `dispatch` method can be inlined to `ip_stack.socket_egress()`.

After lifting `socket.dispatch` into scope, the steps before and after the device usage can be re-encapsulated into package pre-processing `socket.dispatch_before` and the socket post-processing and after sending `socket.dispatch_after`. Thereby we see another refactoring problem. When we split execution steps into different functions. In the original code, the 'packet' i.e. the two representations where used in emit and afterwards to update the socket without any referencing or cloning. This was possible, because they implement `Copy` and can therefor be implicitly duplicated. This still works when we inline the socket.dispatch code because still any processing of the packet happens in a single, now extended stack frame. However, it stops working as we re-encapsulated packet handling before and after sending. The reason is that with that re-encapsulation the frame producing the packet (i.e. the

```
// tcp_socket.rs
fn dispatch(...)
    /*check socket state*/
    if /*a packet can be produced*/{
        let packet = /*produce packet*/;
        neighbor_addr = Some(response.ip_repr().dst_addr());
        let token = device.transmit().ok_or_else(|| {
            Error::Exhausted
        })?;
        inner.dispatch_ip(token, response, None)?;
        emitted_any = true;
        Ok(())
        /*
        if no error occured, update socket state
        using the packet information
        */
    } else {
         return Ok(())
    }
```

Listing 3.5: Packet dispatch function after inlining the `respond` closure

pre-process call) is gone, when we want to send it. This would also happen if we did not re-encapsulate but left the code inside poll. The reason is that in any case the steps will be interrupted by a call to the device i.e. by leaving the current stack frame.

To fix this problem, we introduced a heap-allocated version of the package data type. Also, pre-processing now returns two copies of the packet so that one can be used for sending and one for post-processing the socket. This is a general problem when we want to make sequential code concurrent or even distributed. In the first case, types need to be transferable from one function stack. Ohua cannot recognize, nor can we automatically implement something like copy, clone or serialization for types. Before we assumed that arguments are always passed by value and that serialization in the chosen backend is available for all types used as arguments in the compile scope i.e. all types that are send among nodes later. If we tried to implement refactorings/transformations in the compiler that a) split functions or b) bring code into scope that wasn't before, we need to extend this assumption to all code possibly affected by the transformation. Another notable point here is, that the refactoring must be applied to all types of sockets (TCP, UDP, raw sockets etc.) individually and can not be implemented on trade level.

Refactor token usage

Now we need to eliminate the implicit use of the `device` inside `inner.dispatch_ip(token, response, None)`. The sending `token`, passed to this method contains a reference to the `devices` sending buffer. So the function as it is violates two requirements by using shared references among components and using one component implicitly inside the other. Contrary to packet processing, this method ends with the call to the device, meaning there are no state changes occurring after the sending and except the sending result, nothing depends on the actual device behavior. This means, we can separate the packet preparation inside `inner.dispatch_ip(token, ..)`, from the effective packet sending in the device, making both steps local to their component. For the `ip_stack` we provided a local implementation of the sending token. Like the original token, the `LocalToken` holds

```
pub trait Device<'a> {
    type RxToken: RxToken + 'a;
    type TxToken: TxToken + 'a;
    // ...

    fn send(&'a mut self, timestamp:Instant, packet:Vec<u8>) -> Result<()> {
        let sending_result
            // Request a token locally
            = self.transmit().ok_or_else(|| {
                        net_debug!("failed to transmit IP: {}", Error::Exhausted);
                        Error::Exhausted
                    }).and_then(|token|
                    // copy the packet to the devices sennding buffer
                   token.consume(timestamp, packet.len(),
                     |buffer| Ok(buffer.copy_from_slice(packet.as_slice()))));
        sending_result
    }
```

Listing 3.6: New sending function in the `device`

a reference to a buffer and implements the expected `token.consume` method. Also we replaced the direct call to `inner.dispatch_ip`, by a wrapper function `inner.dispatch_local`. That function instantiates a local sending token, executes `inner.dispatch_ip` and returns the token buffer containing the packet. On the other hand the device needs an explicit sending method, instead of an implicit call via the token. This new `device.send` method basically needs to call the `token.consume` method on a given token, with a given packet. Therefor we do not need to have specific implementations for implementers of the `Device` trait, but can implement `send` directly on the trait. The method `token.consume` does not take a packet, but a closure processing a packet and copying it into a given buffer. Both already happened in the `ip_stack`, so we only need a closure that copies the given packet into the actual device buffer. Finally, with respect to our target architecture there is no particular use in requesting and returning actual tokens to the device any more. In a distributed system, file pointers as contained in the tokens cannot trivially be send among components. Therefor, instead of sending tokens we only request them locally in the device when needed and send `Result<()>` to the `ip_stack` instead. Hence the implementation of the sending function looks as in Listing 3.6:
The functional split between device and `ip_stack` can hardly be derived automatically and in particular statically. To do this, tha compiler would need to understand that i) there are no state changed in the interface depending on the device reaction, ii) that tokens are references to the device, and iii) that the closures passed to the the `token.consume()` function can be split into 'preparing a packet and copying it to a local buffer' and 'copying it to the actual buffer'.

3.1.3.2 Refactor to Message Passing

We inlined the code for `socket_egress` into `poll` and lifted the usages of the `device` into the scope of the `poll` function. This also means that the control and data flow for sending are now explicit in the `poll` function. Figure 3.3 shows the principle structure of the sending loop at this stage. The actual code at this point can be found in the Appendix A.1. As indicated by dashed and solid errors in the graph, the control flow and the data dependencies of the individual steps are not congruent. In particular the currently processed `socket` is needed in multiple steps in the `ip_stack`, but not needed by the `device`.

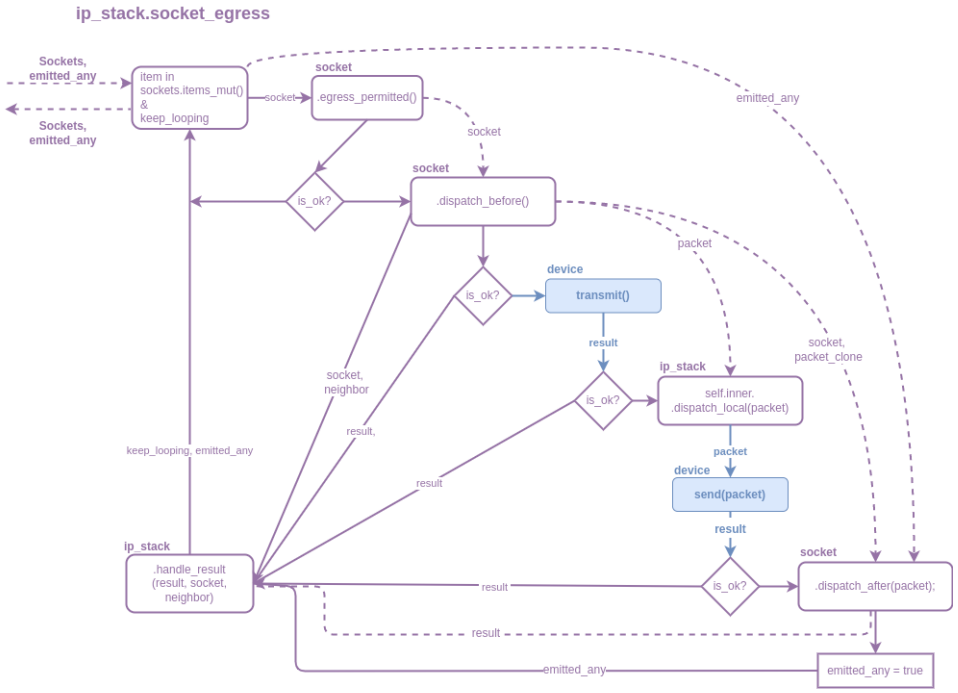


Figure 3.3: Structure of the packet sending loop after lifting all device usages into scope. Solid arrows indicate actual control flow, dotted arrows indicate data dependencies

Refactoring control flow

Currently the sending loop happens entirely inside or above the stack frame of `poll`. That's why it is possible to use data references and resume execution automatically after the `device` is called. We need to refactor the loop such that we can return the frame of the main function when the `device` is called and resume the execution inside the `ip_stack` afterwards. As method calls of the two components are interleaved, we cannot just merge the calls of each component i.e. call the `ip_stack` to get a packet, call the `device` to send it and modify the `ip_stack`s state again based on the `device`s response[2]. This necessarily means, we need to split the execution of `poll` into several method calls and we need to preserve the state of execution and the state of variables inside `poll` from one sub-call to the next.

When we split the `ip_stack.poll` method, or in general any method into separate *sub-methods*, we have two options to transfer data from one call to another. One is pass them explicitly, the other one is to make them part of the state. Serialization is generally expensive and also in terms of security and process isolation, it is desirable to only send data around that is actually needed in other components. Analyzing the data dependencies between the `ip_stack` and the `device`, it is obvious that the only arguments the `device` calls require are the packet to send and, in the actual implementation, a timestamp and the length of the packet. So except for the packet no data needs to leave the `ip_stack` towards the `device`. We will therefore only integrate the packet into the explicit data flow, and preserve the state of other variables in the state of the `ip_stack`.

A common obstacle in both data handling options is the elimination of reference usage. As explained in Section 2.5 Rusts type system will point to a problem, when we try to share references among different

[2]Actually this is an opportunity if state changes in each component are either idempotent or ordered i.e. implemented as Conflict-Free Replicated Data Types. Both is not the case here.

stack frames. By splitting the original `ip_stack.poll` method, we execute code that was supposed to run in one stack frame in multiple frames. This means we cannot simply use the data types of the original program. In particular, regardless if we send data or store them in a state between function call, they must not use references among each other. This is obvious for serialized data, as references serialized and send to processes potentially running on a different machine loose their relation to memory. For the case of data stored in a state, consider the following example from our problem. The sending loop is an iteration over the `sockets`. So at the point, where a packet is produced, there are four references to the same memory location in scope namely i) the sockets ii) the iterator on the sockets iii) the current socket and iv) the packet, which actually is a reference to memory inside the socket until it is finally send. As mentioned before, packets need to be become an owned data type to be sendable to the `device`. Which eliminates the fourth reference. However, we also need to store the state of iteration and the current socket in the state, without references to the sockets. It is in general not possible to return a **`struct`** with fields referencing each other in Rust, because the consistency of references cannot be guaranteed[3]. In particular we cannot have a socket, referencing an iterator, referencing the sockets in our current state, since any of those reference could become independently invalid leaving dangling references.

The problem was solved by implementing an owned representation of the iteration state. As shown in, we integrated the current state of `ip_stack.socket_egress` as a separate, optional field to the `ip_stack`. Instead of saving an iterator and a reference to the current socket, only an index of the current socket in the socket set is stored. The socket itself must be retrieved in ever step using it. This also implied that we need to rewrite the iteration on the socket set. Instead of using the given `Iterator` implementation, we implemented iteration over the size of the socket set, starting at the currently saved index. We did the same for the other intermediate results of the original method `neighbor` and `packet` and `address`. The resulting definition of the `ip_stack` data type is shown below.

Like the previous refactoring, this step can not be entirely automatized, because the compiler can not derive owned or serializable representations from arbitrary data types.

```
struct Interface<'a> {
    ..,
    egress_state: Option<EgressState<'a>,
}
struct EgressState<'es>{
    sockets_during_egress: Option<SocketSet<'es>>,
    current_index: Option<usize>,
    current_neighbor:Option<Address>,
    current_presend_packet: Option<IpPacketOwned>,
    current_postsend_packet:Option<IpPacketOwned>
}
```

Now the control flow needs to be refactored, such that the device is called in the main scope and `device` and `ip_stack` have each only one method call visible to the compiler. This method will be *entry point* of both states and internally dispatch the call to the required code. We will call the method `process_call`

[3]Actually there are ways to achieve memory consistency across function calls in Rust using e.g. the `std::pin` module. However, they are not applicable to other languages so we do not consider them here.

and start by defining it on the `device`.

Intuitively `process_call` would be a higher-order function. Specifically, instead of calling the `device` directly, `poll` would return the functions `transmit` or `send` including the necessary arguments. The device would then be called with `device.process_call(fn, [args])`. However, higher-order functions are not a viable solution here. This is for two reasons. Firstly we envision to send the functions to be called by `process_call` from one component to the other. But function references are, with some exceptions, not serializable and sendable in distributed scenarios. Secondly we also need to pass the arguments for those called functions to `process_call` so we would need to derive a common sum type to represent all possible function arguments.
The solution to this is called defunctionalization. It is a known transformation used in compilers to transform higher-order functions [41]. Instead of sending function references, a sum data type representing all possibly called functions is defined. In Rust we can use **`enum`**s for this purpose. This **`enum`** data type also allows us to integrate the arguments called per function so the **`enum DeviceCall`** looks as follows for the sending loop:

```
#[derive(Debug, Eq, PartialEq, Clone)]
pub enum DeviceCall{
    Transmit,
    Consume(Instant, Vec<u8>),
}
```

We merge control flow and data flow here so the return type will be the union of the return types of all functions that might be called, wrapped in the respective next step of execution. In case of the `device`, the next step of execution will be a the next part of the `poll` function to be executed. Therefor we define `device.process_call` as follows:

```
pub trait Device<'a> {
    // ...

 fn process_call( &'a mut self, call: DeviceCall) -> InterfaceCall
    {
       match call {
         DeviceCall::Transmit
             => InterfaceCall::InnerDispatchLocal(self.transmit()),
         DeviceCall::Consume(timestamp, packet)
             => InterfaceCall::SocketDispatchAfter(self.send(timestamp, packet)),
    }
}
```

In principle, each `device.process_call` now returns a continuation in the `ip_stack`.

For the `ip_stack`, the implementation of `process_call` and the **`enum InterfaceCall`** is slightly more involved. Contrary to its implementation on the `device`, the `ip_stack.process_call` method will need to call sub-sections of the the `ip_stack.poll` method. We will consider the simplified pseudocode version shown below for now, to illustrate the principles applied. In Listing 3.7, borders of basic blocks, i.e. points the control flow could jump to are marked and given names, because they are needed for the transformation.

```
impl Interface {
    // 1st jump => InitSimplePoll
    fn simple_poll(&mut self, sockets: &mut SocketSet, device: &mut Device) -> bool {
        let mut something_changed = false;
        let mut result: SmoltcpResult<()>;
        // Jump here 2nd => StartSendLoop
        for socket in sockets {
            // Jump here 3rd => GetNextPacket
            let next_packet = self.maybe_get_packet(socket);
            if next_packet.is_ok() {
                result = device.send(next_packet);
                something_changed = result.is_ok();
            } else {
                result = next_packet.as_err()
            }
            // Jump here 3rd => HandleResult
            self.handle_result(result)
        }
        something_changed
    }
}
```

Listing 3.7: Blocks of the `ip_stackpoll` function that need to be directly callable after refactoring

Now the transformation from `ip_stack.simple_poll` to `ip_stack.process_call` would proceed as follows. First the basic blocks are identified. From those blocks, a corresponding **`enum InterfaceCall`** is defined representing each of them. If a block uses results from the `device`, this results are part of the constructor of its `InterfaceCall` representation. For our simple example this would yield the following **`enum`**:

```
enum InterFaceCall<'a> {
    InitSimplePoll(SocketSet<'a>),
    StartSendLoop,
    GetNextPacket,
    HandleResult,
}
```

The `ip_stack.process_call` is again essentially a **`match`** expression on the given call. The arms of the matches wrap the respective parts of `simple_poll`. The next modification we need, is to augment every such arm with statements to restore the original environment of each block inside the match arm. We stored the variable state inside the state of the `ip_stack` as described before, in this trivial example we assume to set them directly as filed of the `ip_stack`. However, the code in the branches might still use references to the variables and we need to respect rusts borrowing rules, so we cannot generally replace any direct variable use `f(x)` by `f(self.x)`. So we add statements to get and set the stored variables in each block before the code and before each **`return`** statement respectively. Finally we replace calls to the device by statements returning the respective `DeviceCall` and jumps to other blocks by the corresponding call of `ip_stack.process_call(next_block_call)`. The actual return statement of `simple_poll` remains unchanged. So the return type of this simplified `ip_stack.process_call` is `Either<DeviceCall, bool>` and is shown in Listing 3.8.
The actual implementation of `ip_stack.process_call` is obviously more complex, but was build in the same transformation steps.

```
fn process_call(&mut self, call: InterFaceCall) -> Either<DeviceCall, bool> {
    match call {
        InterFaceCall::InitSimplePoll(sockets) => {
            // set variables something_changes, result, sockets and index
            return self.process_call(StartSendLoop)
        },
        InterFaceCall::StartSendLoop => {
            // get socket and socket_index
            if socket_index+1 < sockets.size() {
                let socket_index = socket_index+1;
                // set sockets and socket_index
                return self.process_call(GetNextPacket);
            } else {
                // set sockets and socket_index
                return Either::Right(self.something_changed);
            }
        },
        InterFaceCall::GetNextPacket => {
            let next_packet = self.maybe_get_packet();
                if next_packet.is_ok() {
                    return Either::Left(DeviceCall::Send(next_packet))
                } else {
                    self.result = next_packet.as_err();
                    return self.process_call(HandleResult)
                }
        }
        InterFaceCall::HandleResult => {
            self.handle_result();
            return self.process_call(StartSendLoop)
        }
    }
}
```

Listing 3.8: New entry function of the `ip_stack`

3.1.4 Refactoring Packet Receiving – Completing the `poll` loop

Just like sending, receiving is triggered in the `ip_stack.poll` method. The main aspects of `ip_stack.socket_ingress()` are depicted in Listing 3.9. In a while loop, the `ip_stack` tries to receive packets from the `device`. In this case packets are represented by a token pair. The receive token `rx_token` contains a buffer holding the received packet, the send token `tx_token` may be used to directly return an answer to the `device`. During initialization, the transport medium used by the `device` is saved in the `ip_stack`. By matching on the current transport medium the method to unwrap the outer most layer of the packet is determined, in this case `process_ethernet(sockets, &frame)`. Using the reference to the sockets the function determines which socket the packet should be dispatched to. Next packet is processed by the socket and copied into its receive buffer. From there it can later be retrieved and passed to the `store`. The socket may also produced a direct answer, for example as part of the TCP protocol. If so this answer is dispatched as discussed before for the sending loop using the `tx_token` as a reference to the device. So in essence the `device` is used three times here. First time explicitly for receiving, the second and third time implicitly by accessing its receiving and sending buffer via the tokens.

```
fn socket_ingress(&mut self, sockets:&mut SocketSet) -> bool {
    // ...
    while let Some((rx_token, tx_token)) = device.receive() {
       if let Err(err) = rx_token.consume(inner.now, |frame|
          match inner.caps.medium {
              Medium::Ethernet =>
              match inner.process_ethernet(sockets, &frame) {
                  Ok(response) => {
                        processed_any = true;
                        if let Some(packet) = response {
                            if let Err(err) = inner.dispatch(tx_token, packet) {
                            /*log Send Error*/
                            }
                        }
                        Ok(())
                    }
                },
                _ => /*handle  other Medium types*/
            }) else {/*log Handling Error*/;
            }
        processed_any
    }
```

Listing 3.9: Simplified code of the `Interface.socket_ingress()` method

Again we need to eliminate the use of common references between `ip_stack` and `device` and integrate the receiving control flow into the `process_call` logic of both components. So we first eliminate the use of reference tokens. A packet can no longer be received via a reference to the `device`, hence we implement a new receiving method on the `device`. This new method `device.receive_token_free` first executes `device.receive`. Receiving might or might not succeed, so it needs to return an `Option<Tokens>`. If token tuple is returned, an owned buffer is allocated and `rx_token.consume`

```
fn pseudo_ingress(&mut self, device: &mut Device) -> bool {
    while let Some(packet, send_permission) = device.receive_tokenfree() {
        let local_tx = /*init LocalToken*/;
        match inner.caps.medium {
            /*process &packet and local_tx as tokens before*/
        }
        if local_tx.len != 0 {
            let packet = local_tx.take_buffer();
            if let Err(e) = device.send(packet) {
                /*log Send Error*/
            }
        }
    }
}
```

Listing 3.10: Receiving function in the `ip_stack` using a local token

is called to copy the received data into that owned buffer. Instead of returning an `rx_token` and a `tx_token`, `device.receive_tokenfree` then returns the received data and simple result representing the `tx_token` (i.e. `Ok(())` or `Error`). Consuming the `rx_token` might return an error that needs to be logged in the original code, however this error can only occur during processing the `ip_stack` so `device.receive_tokenfree` does not need to additionally the result of consuming.

Inside `ip_stack.socket_ingress` the usage of the `rx_token` can now be replaced by simply passing a reference to the token to `inner.process_ethernet(sockets, &frame)`. If a response is returned, we use the same mechanism as in the sending loop before. A local token is passed to `ip_stack.inner.dispatch` and the `device.send` function is used to explicitly send this owned token buffer to the device. After this refactoring, the core control flow between the two components looks basically as in Listing 3.10:

There are now two explicit, stateful usages of the `device` and we can apply the same procedure as with the `ip_stack.socket_egress` function, to integrate the basic blocks of `ip_stack.socket_ingress` into the `ip_stackstack.process_call` matching. The method itself is divided into three additional `InterFaceCalls` i) `InitIngress` to trigger the ingress call including the first receive call to the device, ii) `ProcessIngress` handling either a received packet or ending ingress and iii) `LoopIngress` logging the device result and calling the `device.receive` again. For the `device` we also needed to add two calls. One is obviously `DeviceCall::Receive`. The second one is needed, because the call `DeviceCall::Send` is now used in two different control flow states of the `ip_stack`, once during egress and once during ingress. So we need a way to distinguish in the `ip_stack` if the control flow should continue in the egress or the ingress loop after a `device.send` call. There are at least three options to handle this, The first is to create a new `DeviceCall`, the second is to save the current state in the `ip_stack`and add an additional dispatch based on this state, the third one is to make the current state another value in the data flow and have the device decide which `InterfaceCall` to return after sending. We decided to use the third option and augmented the `DeviceCall::Send(Packet)` with an additional parameter `enum InterfaceState` being either `Ingress` or `Egress`. The device would now also pattern match on this parameter and return the next call to the `ip_stack` accordingly.

```
let mut ip_stack_call = InitPoll;
loop {
    let device_or_app_call = ip_stack.process_call(ip_stack_call);
    if calls_dev(&device_or_app_call){
        ip_stack_call = device.process_call(device_or_app_call);
    } else {
        let socket = sockets.get_mut(socket_handle);
        /*first socket state handling */
            let outbytes = store.handle_message(input);
            socket.send_slice(outbytes);
        /*second socket_state_handling 2*/
    phy_wait(/*...*/);
    }
}
```

Listing 3.11: Structure of the server loop before socket handling is moved into the `ip_stack`

Now there is one missing element in the basic loop. In Section 3.1.2 it was described that the socket handling, i.e. setting sockets to listening state, calling the receive function on the socket etc. had to be moved into the `ip_stack` code. To simplify previous explanations we ignored this so far. If we left the socket handling in scope, the main loop would be structured as shown in Listing 3.11:
We can see that socket handling happens in two parts. One directly after the `ip_stack` returned from polling, the second after the `store` is called. Therefore, as socket handling was moved into the `ip_stack`, only one new `InterfaceCall` was introduced. The first part of socket handling was directly attached to the match arm of the last `InterfaceCall` during polling. Instead of only returning the polling result, the closure ending poll now also returns the request if the socket received one. The second part of socket handling was moved to a separate `InterfaceCall`, capturing the result from the `store`. This also meant that waiting before the next poll i.e. the `phy_wait` call now happens between two `ip_stack` calls and needs to change its position in the loop. In transforming `phy_wait`, aspects had to be taken into account that have not been addressed so far. Therefore, this will be considered separately in the next section.

3.1.5 Waiting in M³

Now we need to refactor the `phy_wait` call. This function is basically a wrapper function to the system call `libc::select`. It accepts a file pointer and a waiting duration and halts the current thread until either the file becomes available or the waiting duration is exceeded. The file pointer is a pointer to the system file the device uses to write the packets to. The maximum waiting duration is determined by the `ip_stack` based on the current state of sockets. We are facing two problems with this function. One is that we need to integrate it into the refactored control flow. The second is that it is using a system call and a file pointer. Both file access and system calls are operating system specific, and therefore may be implemented differently in m3 or any other target architecture.

Functionally, three things are realized in the `phy_wait` call. Determines the maximum waiting time for the sockets is obviously a reading operation on the state of the `ip_stack` and should be integrated in `ip_stack.process_call`. Determining the file pointer availability in a distributed setting needs to happen in the component that holds the pointer, so we will need another call to the device. Finally

```
loop {
    let device_or_app_call = ip_stack.process_call(iface_call);
    if is_device_call(&device_or_app_call){
        let iface_call_or_wait = device.process_call(device_or_app_call);
        iface_call = maybe_wait(call)
    } else {
        // call the store to answer messages
    }
}
```

Listing 3.12: Final server loop structure

waiting is a system operation. The component that should wait is the interface. However, we need to implement waiting in an m3 specific manner and therefor we either have to do it in the main scope, or we have to import the according m3 API into the `interface` module. The clearly preferable option is to wait in the main scope over mixing component internal logic and runtime logic. So in essence wwe refactor the `phy_wait` call to another loop between the `ip_stack`, the `device` and the main scope

As waiting should always happen immediately after 'loading' the sockets with answers from the store, we do not need to introduce another `InterfaceCall`. Instead we rewire the internal logic of `ip_stack.process_call`. So far we just ignored waiting. After the sockets are loaded with messages from the store in `ip_stack.process_call(AnswerToSocket(/*messages*/)`, we would directly proceed calling `self.process_call(InitPoll)` again. Now instead the waiting time for the sockets is calculated as before in the main loop using the `self.poll_delay()` function. Then the calculated duration is send to the device using a new `DeviceCall`.

To process the new `DeviceCall::NeedsPoll(Duration)`, we define a method `needs_poll` on the device trait. M^3 implemented its network stack with smoltcp, but provides altered versions of the devices. Those devices provide the `needs_poll` method to implement waiting in the main scope, so this refactoring mimics their behavior while allowing us to run our refactored smoltcp version also on standard Linux. The method accepts a duration and returns a boolean indicating if the device is available again. In the concrete implementations of the device trait we can again use `phy_wait`, to maintain original behavior while we do not run on M^3 .

Finally we need to alter the general control flow. Waiting has to happen in the main scope in M^3 . So after calling the `device` with `DeviceCall::NeedsPoll`, the `ip_stack` should not be invoked directly as before but instead a waiting function is called, mimicking the behavior of M^3 waiting implementation. This means a call to the `device` can now either return a call to the `ip_stack` or an instruction to wait, containing the information M^3 will require for waiting. Consequently in the main scope a new case distinction is needed. If the `device` returned an `InterfaceCall`, it is directly forwarded to the `ip_stack`, otherwise the waiting instruction is executed by a placeholder and and the `InterfaceCall` to restart polling is generated. To keep control flow simple in the main loop, this case distinction is wrapped in a function `maybe_wait`, yielding a control flow as shown in Listing 3.12.

3.2 Adaptations in Ohua

Although Rust is a strongly typed language, type annotations for local variables are often unnecessary, since they are automatically derived from the type inference of the rust compiler. In contrast, definitions of types, i.e. structs, enums and functions, are the basis of type inference and must be annotated manually/by the programmer. Since Ohua produces new code in the backend, it is not enough to transfer existing annotations of the input to the output. In particular, the communication channels of the produced tasks require type annotations, since these cannot be completely derived by the Rust type inference even in the shared memory scenario. Also in view of a later extension to fully distributed systems, distributed compilation of individual components or the use of languages with less powerful inference, it is important not to rely on the type inference specific to the compiled language.

So where do we get the type information we need? One important observation is that Ohua does not create new types. The control functions Ohua inserts into the data flow graph are mainly there to pass on existing intermediate results of the original program. That is, the types of these functions that is, of their input and output channels can be derived from the input code. For this we need a) a type extraction from the input program and b) a type propagation which propagates the corresponding types through the representation of the data flow graph. Both were basically available at the beginning of the work. However, the existing implementation had problems or was not fully functional. This means that some types had to be annotated by hand in the output code. The following sections describe how type extraction and type propagation worked and which changes were necessary to achieve the desired functionality.

3.2.1 Type Extraction

So the first functionality we had to address was the TypeExtraction in the frontend integration. Until now this was a two step process. In a first pass two kinds of data were extracted from the input module. One was the algorithms i.e. the Rust functions that were to be compiled. Those were translated to an internal representation of the supported Rust subset as described in Section 2.1.1. The second structure kept track of imports defined in the module.

The second pass was needed to annotate types to function names. As each function call
`let z = someFun(x, y);` might become an independent node, the compiler needs type annotations for `x` and `y` to later annotate the channels for sending those variables among independent tasks. To do so first the function names called in the parsed algorithms were extracted. Then all files defined in the imports where scanned for function definitions. From this information, a hashmap was built over all function names and the extracted function types. Finally, in a further traversal over the input code, this hashmap was used to lookup function types and annotate them in the input code. This procedure had some disadvantages, namely :

- The entire compile scope, including for example the standard library, had to be available for the compiler to find and process. This introduced path dependencies of the compiler and notably excludes the import of compiled libraries in other languages e.g. libc, which is critical in our case.

- We had to restrict the entire compile scope and imported libraries to syntax the compiler could understand. This has previously been addressed by re-implementing some required libraries in a simpler form.
- We had to keep track of name spaces and aliasing for all functions
- We could not support 1. generic type parameters in function definitions and 2. overloaded function definitions.

The main point of concern was really the need to parse the whole scope and therefore to have all libraries used available and compatible to Ohua supported syntax. Therefore, we chose to change the source of type information. Instead of collecting function signatures from the scope and typing functions globally for the complied module, we now type each call site according to the local context. Remember, we need to type the input parameters for each function call inside an algorithm. Those parameters can be:

- global constants, in which case Rust requires a type annotation
- input parameters of the algorithm, in which case we know their type from the algorithm signature
- local variables bound in the algorithm, in which case we now require the programmer to provide type annotation

So for most syntax constructs, we can derive the type information needed from the local context. This requires the programmer to annotate types manually in local assignments, where it would not be required by Rust itself. Listing 3.13 shows an example, where additional local bindings are required. In the example code, to be able to type the function call to `h(e)` we need an additional binding statement as it is not possible to type annotate a loop pattern currently.

```
fn test(i:i32) ->  {
    let s = State::new();
    for e in range_from(i) {
        let r = h(e);
        s.some_method(r);
    }
    s
}
```

```
fn test(i:i32) -> () {
    let s:State = State::new();
    for e in range_from(i) {
        let e1:i32 = e;
        let r:i32 = h(e1);
        s.some_method(r);
    }
    s
}
```

Listing 3.13: To extract type information for function call from the local context we require the programmer to annotate the according types to local variables. As shown in the right code example it is sometimes also necessary to have additional binding statements to annotate every relevant variable i.e. every input to a function call.

The main change, required to implement this solution was threading a monadic context through the complete process of transforming the input code to the frontend representation. In particular also through the first step of this process, where the Rust code is mapped to a subset of Rust supported by Ohua. These context keeps track of variable bindings and according types in the current scope. In the outermost scope i.e. the global level of the input code, this context is pre-filled with constant definitions, including all global constant definitions parsed before the actual algorithms.

The conversion of algorithms is also monadic process and the initial context contains the names and types extracted from the global scope. For each algorithm the parameter names and their types are parsed from the signature and added to the context. Upon parsing the body of the algorithm each right hand side of a let binding[4] is checked for type annotation and registered in the local context if annotated properly. Unannotated bindings will yield an error at this point. AS shown in the example in Figure 3.13 this might require some additional local assignments, when local variables result from pattern binding.

Now, whenever a function call is converted, the function type is derived from its arguments. In the case of a stateful call, this also includes the called object. If the arguments are variables, the types are obtained from the context. If the variables are not in the context, again an error is thrown. For supported literal arguments (currently integers, booleans and strings) the types are derived automatically. Obviously, this limits the accepted input syntax in that only literals and variables are valid as direct function arguments. Still the advantage of being able to use the entire Rust syntax again in the imported libraries outweighs this in our opinion. As function types now depend on the types of local variables we added test cases to the regression test suite to ensure proper typing, when local scope variables shadow names from outer scopes. Notably name shadowing is currently only supported for loop local scopes. Ohuas renaming algorithm and could therefore not be tested.

3.2.2 Type Propagation

The second aspect arises in the core compiler. When Ohua generates a Data Flow Graph, it introduces control functions e.g. to guard branching or collect results of a loop (see [18] for further details). Some of these functions are only present in intermediate representations because node fusion implemented downstream in the compiler may integrate them into bigger nodes. Some however occur as separate tasks in the final program. In particular for the later kind, proper type annotations are essential but it is sensible to provide them for any such function if possible, to reduce the assumptions among the steps of compilation. Also the transformation of code to SSA form introduces new variable names that need to be typed.

Control function by their nature do not introduce new types. So it is possible to infer most of their input and output types from the host-language types parsed in the frontend. The code example in Listing 3.14 shows a Rust function and its last representation in the compilers DFLang representation. Marked in **bold** we can see the functions Ohua introduced to control the dataflow in the final program. We can also see that one of those functions, namely $\mathbf{smap}$ is not preceded by the namespace marker $\mathbf{ohua - lang/}$. This is because most of the control functions are currently not represented by own constructors of the function representation in DFLang. They are represented internally just the same as host-language function calls and only recognized in pattern matching upon their names. In contrast $\mathbf{smap}$ is already implemented as a separate constructor.

[4] We currently only support variable or tuple patterns

```
fn test(i:i32) -> () {
  let s:State = State::new();
  for e in range_from(i) {
     let e1:i32 = e;
     let r:i32 = h(e1);
     s.gs(r);
  }
}
```

```
let s_0_0_1 =
    ohua.lang/unitFun(State/new_state, ()) in
let a_0_0 = /range_from ($i) in
let (d_1, (ctrl_0_0, ctrl_0_1), size_0) =
    smapFun(a_0_0) in
let s_0_0_1_0 =
    ohua.lang/ctrl(ctrl_0_0, s_0_0_1) in
let lit_unit_0 =
    ohua.lang/ctrl(ctrl_0_1, ()) in
let r_0_0_0 = /h (d_1) in
let (_, ) = /gs [s_0_0_1_0] (r_0_0_0) in
let d_0_0 =
     ohua.lang/unitFun(ohua.lang/id, lit_unit_0) in
let x_0_0_0 =
    ohua.lang/collect(size_0, d_0_0) in
let c_0_0 = ohua.lang/seq(x_0_0_0, ()) in
c_0_0
```

Listing 3.14: Example of a Rust input function and its last stage in the core compiler representation DFLang. Functions in **bold** are control functions, introduced during compilation that need to be type annotated.

Basically, the TypePropagation works as follows. Remember in the frontend we annotated the argument types for the called Rust functions, in the example in Listing 3.14 the function calls `State::new_state()`, `range_from(i)`, `h(e1)` and `gs(r)`. Using this information the type propagation happens in a bottom-up traversal over each compiled algorithm. Due to this bottom-up processing each use of a variable is processed before its assignment. That means in our example, the function call `(_, ) = /gs [s_0_0_1_0] (r_0_0_0)` is processed before the assignment `r_0_0_0 = /h (d_1)`. As the function call is processed its argument types are used for two things 1. update the type field of the variables used in the call and 2. update the context to contain the associations between the variable names and the according function type arguments.

Contrary to Rust function calls argument types of Ohua control functions are not annotated at this point. However, we can always tell their output type from the input, because they only guard data flow and do not calculate results themselves. For example the collect control function is introduced to collect the results of a loop. We know its signature has to be collect :: nat− > A− > [A], to collect a given number of arguments of type A before returning a list of type [A]. If the output list is used by another function downstream, we already know the type A from the context and can completely annotate the variables in statements using collect.

Now there were two problems with the existing implementation of the TypePropagation. The first rather trivial one was that several control functions where not or incorrectly processed. The second problem is illustrated in the code example. It is possible that a graph ends in one or more control functions. In this case there output is not used by any typed Rust function, so we could not type there arguments and returns in the bottom up pass. Due to this problems, the existing TypePropagation was only partially functional. While the former TypeExtraction mechanism just complicated or restrained the assembly of proper input code, missing TypePropagation functionality actually leaded to non functional code, in the

sense that type annotations had to be made manually in the output code after compilation.

To fix this issue we have made two main changes. First, we have extracted the return type for all algorithms, i.e. all compiled functions in the frontend. This is now passed as an additional parameter through the compiler pipeline and is available in the type extraction. In the example code in Listing 3.14 the return value is `c_0_0`. Given this type information we can now also annotate Ohua control functions whose output is not used by an annotate Rust function in the bottom-up pass. The second change was obviously to fix or add type propagation for previously wrongly typed control functions. Finally all tests where adapted to expect correct typing of the output code.

One problem that has not been addressed in this work is that the control functions are not represented by separate constructors in the DFLang. This means that they can only be distinguished from normal Rust functions by their function name, which is error-prone and difficult to maintain. So in the future the control functions should at least be mapped in their own constructors. Also arguments with known type, like the first argument of the function collect :: int− > A− > [A] should optimally be enforced by the type system.

3.2.3 Destructuring Higher Arity Tuples

Destructuring of function output was limited to to flat tuples of two elements. As shown in the Listing 3.15 below, a function output consisting of more than two elements would have to be destructed in steps. This was obviously inconvenient, but more importantly, the added destruction step, as other function calls would have created a separate node in the derived DFG. This means that processes or threads were created for these unnecessary destructuring nodes, and the corresponding data was unnecessarily serialized and deserialized via these processes. In this smoltcp case study the problem is even more pronounced. Remember the goal is to not send the stateful components from one process to another at all. However, the recursive loop of preparing, sending, receiving, and processing data takes all three components as well as the data as arguments. So the limitation of destructuring would have required us to introduce destructuring steps in the input code and prevented state locality in the final graph.

```
let (x, y, z):(A, B, C) = actual_fun(); let (x, yz):(A, (B, C)) = actual_fun();
                                        let (y, z): (B, C) = invented_destruct(yz);
```

Listing 3.15: With destruction being limited to two elements, programmers needed workarounds as additional destruction steps to use function outputs of more that two variables

Therefore, we had to extend the support for tuples in three ways. Firstly we made tuple destructuring representable in the backend language. In particular tuples where represented there as expressions Tuple(e1, e2), where the expressions e1 and e2 where either literals or variables. Tuple indexing was implemented as dedicated expressions Firstbnd and Secondbnd, where the binding bnd was the name of the indexed variable, e.g. First"myList" would be converted to `myList[0]` in the Rust backend.

Now tuple expressions are represented as lists of literals or variables Tuple[EitherLitVar] and indexing is represented via the more general term Indexingbndnum, representing indexing of bnd at the natural

number index num.

4 Related Work

4.1 Flexible OS approaches

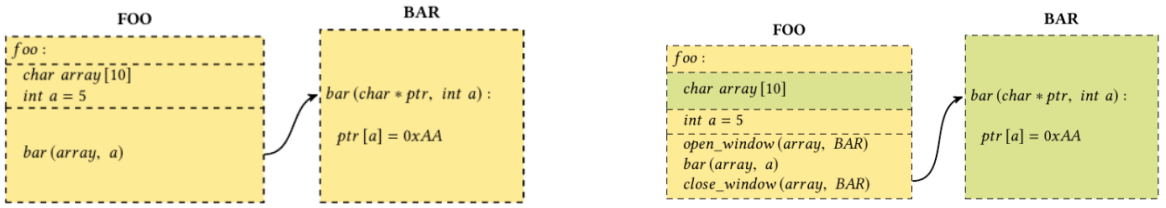
The problem of overcoming disadvantages and potentially combining advantages of microkernel-based operating systems and unikernels has been addressed before. For unikernels disadvantages were a lack of isolation among components and the necessity to adapt to the component the library OS provides. Microkernels on the other hand provide strong isolation but only at the cost of significant overhead for IPC calls and again the necessity to adapt the existing code not to provided components but to communication primitives provided by the kernel. To avoid unnecessary overhead, programmers may also need to refactor the general structure of their code to minimize inter process communication.

One example approach is CubicleOS [44]. It provides three main, new abstractions to overcome the problems of the isolation vs. overhead trade-off. Those abstractions are

1. *cublicles* used to define memory-isolated processes (components)
2. *windows* used to define temporary memory sharing among trusted components, and
3. *cross-cubicle calls* used to implement control flow integrity among *cubicles*

Memory isolation is implemented using Intel's Memory Protection Keys (MPK)[25]. A mechanism implemented in the ISA that manages access rights to virtual page tables based on keys assigned to processes. The current implementation of CubicleOS is based on the Unikraft library OS[29]. The programmer has to specify the components that should become *cubicles*. During the build process, function call among *cubicles* are identified. To enforce isolation the build system of CubicleOS generates enveloping functions for those calls. These enveloping functions called *cross-cubicle calls* implement the context switch between *cubicles* at runtime. Applications using CubicleOS can run on standard Linux. To enforce memory isolation, CubicleOS comes with two runtime components one loading the components with according memory rights, one managing memory access rights.

The main adaptations the programmer is required to make are a) using Unikraft components b) defining the *cubicles* and c) defining exceptions from the memory isolation. Exceptions are needed to improve performance and lower the overhead of context switches, when isolation is not desired. Those exception cases are either whole *cubicles* or just data structures shared for particular calls among components. *Cubicles* that are used frequently and are trusted can be declared 'shared' meaning that there will be no context switches upon calls to any of their functions or usage of static constants. Data structures can be shared using CubicleOSs API for *window*s, which enables the programmer to specify memory locations and sizes and the coded sections for which they should be shared. An example of this API is shown in Figure 4.1, adapted from the original publication.



(a) Original code of application FOO calling library BAR

(b) Annotated code using Cubicles *windows* to share memory after compilation

Figure 4.1: To use Cubicle the developer basically needs to surround calls to other components, in this case BAR with *windows*. Cubicle will derive isolated components and their (legitimate) interaction

The aim of FlexOS [32] is to provide an easy way to exchange isolation primitives for existing code without (extensive) rewrites. Like CubicleOS it is based on an adapted version of the Unikraft library system. The two main primitives the FlexOS API provides are *abstract gates* and *abstract shared data*. In order to compile code with FlexOS, the programmer must replace all function calls between components with *abstract gate* definitions and all shared memory areas with *abstract shared data* definitions. These adjustments have also been made in the adapted system libraries. In addition the FlexOS build system needs a configuration file, which defines among other things the isolation mechanism and the division of the components. During compilation, the abstract definitions are replaced by concrete mechanisms of the target memory isolation techniques, which are MPK, Software Guard Extensions (Intel SGX) and Extended Page Tables [25].

In comparison to our approach, FlexOS and CubileOS solve a very similar problem with very similar constraints for the programmer. She has to define components and their communication explicitly and the adaptation to a concrete architecture is only possible through compilation, because she has to select from suitable system calls and libraries in the input already. In contrary to those systems, we do not provide an API to explicitly allow data sharing via references. It is possible to use reference sharing in Ohuas input programs, as only explicit reference passing is excluded. However Ohua will tread arguments passed by reference the same way as arguments passed by value. So if the resulting program is functional and if pass-by-reference is any more efficient than pass-by-value depends on the target architecture. Which also means we could not effectively use FlexOS or CubileOS as backend integrations.

4.2 Compiling to State Local Programs

Of course, many of the problems and approaches that emerged in the implementation are not new. Denationalization as a concept for mapping higher-order functions to serializable data types was first presented in the work of Reynolds [41]. Building on that concept, as well as Danvy [13] discussed how defunctionalization and refactoring to continuation passing style (CPS) can be used to transform programs with structural operational semantics (including interruptions and errors) to reduction semantics. The author integrates their results with previous work yielding the transformation based equivalence graph shown in Figure 4.2. Given those results we can explore and maybe better describe in future what our transformations should be, for example it would also be conceivable to defining stateful objects as abstract

machines within the DFG. In a subsequent work Danvy and Millikin [14] present *Refunctionalization* as the left inverse of Defunctionalization. This transformation can be applied as an intermediate step, if programs are not directly amenable to defunctionalization. As the authors describe it "A program can fail to be in the image of defunctionalization if there are multiple points of consumption for elements of a data type or if the single point of consumption is not in a separate functionÂ´Â´. Considering the initial structure of the `poll` function, the transformation described by Danvy and Millikin might also be necessary to formalize the findings in this work.

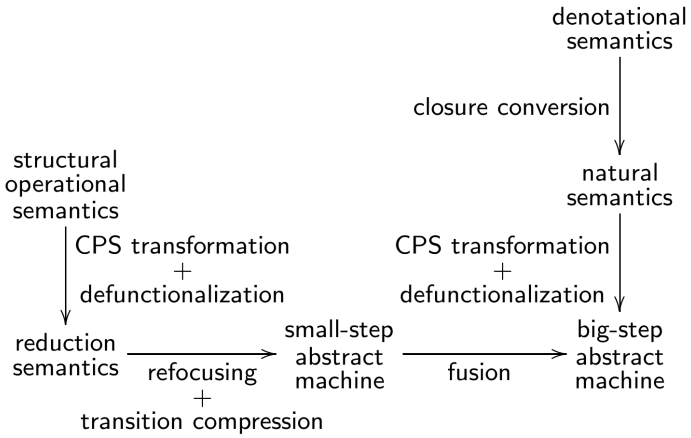


Figure 4.2: Graph representing the inter-derivability relation of different abstract models for computation, taken from Danvy [13]

The paper *Automatically Restructuring Programs for the Web* [19] also applies refactoring to CPS followed by defunctionalization. They target the transformation of local, interactive programs to web CGI programs. As with the `ip_stack.poll` function in our case, the control flow in the source program is continuous in or above the stack frame of the server functions, while the target setting requires explicit handling of both the control flow and the stack state upon leaving an reentering the server function. The paper presents a prototypical implemented automated transformations, to turn a local interactive program into a CGI program. Interestingly in Scheme, the implementation language used, continuations are first-class members and could be stored and reapplied to stack directly assigning each a specific URL, such that the client request can directly address the continuation. Problem was that this was a) language specific and b) had lots of overhead for a distributed garbage collection of stored continuations. So the solution was to embed the control flow into the data flow using defuncionalized representations of entry points. To handle stateful objects, i.e. in general any variable that is reassigned in different invocations of the server, they use the Scheme concept of boxes.

This approach is very similar to what has been done in this work, e.g. by making part of the state of the interface. They must be defined globally, are loaded on each call (from memory or a cookie) and allow all sub-functions of the original server function to access the stateful variable. Considering the server-client setting, the authors also comment on security considerations. The transformations made de facto lead to control flow being integrated into the data flow and, in the case of a web application, can be changed by a malicious user. Even if we aim at a different application scenario, we should evaluate which parts of the control flow actually need to be integrated into the data flow. The security measures

proposed by the authors mainly concern the cryptographic protection of the communication and are in this respect possibly a question for further architecture integrations in Ohua.

The problem of managing stack based data in event-driven and cooperative programming has also been addressed in [7]. Ohua doesn't explicitly aim at event-driven programs. However we have seen that the principle problem of shifting from automatic stack management to a control-flow "re-entering", necessitates to either store and retrieve the execution state explicitly or send parts of it along the data flow, or even passing respectively. The authors discuss necessary refactoring steps to transform single-threaded programs to either a multi-threaded, preemptively scheduled or even-driven versions. Regardless of the final form, they described the necessity of "Stack Ripping", i.e. manually handling the information formerly based on a single function stack. While they use a different vocabulary, the core points of the transformations are actually the same as in the works cited before, facing four main requirements for the implementation resulting from "Stack Ripping"

1. lifting closures: Parts of functions interleaved with I/O actions or calls to other components need to become language-level functions.

2. function scope: As more than one function now represent what was one function before the original environment of the code in each sub-function must be manually preserved

3. automatic variables: variables that where allocated on the stack, but need to survive multiple function calls now, must be allocated on the heap

4. control structures: Structures like loops or branches lead to additional entry point functions

Further they describe how in Windows, fibers and threads are used to implement interactions between applications using manual or automatic stack management respectively. The identified requirements are very much in line with the necessary refactorings we identified.

4.3 (Automatic) Memory Isolation

Besides virtualization and the direct implementation of microkernel-based systems, there are also other approaches to address the security problem of the lack of isolation. One such system is *Hardware-Assisted Kernel Compartmentalization*(HAKC)[37]. The authors exemplify the problem for Loadable Kernel Modules (LKM), which are themselves not a part of the Linux kernel but loaded and executed in kernel space. With many of them providing audio and media processing and drivers, they are a notorious entry point for system compromise exploiting local bugs. In particular they describe an example CVE, where the exploit neither violates memory safety nor control flow integrity and could hence only be prevented by compartmentalization of the modules involved and a defined interfacing to the kernel. HAKC provides an API to define components and a hardware-based runtime system to enforce in-kernel isolation of the defined components. This API allows the programmer to assign compilation units, files or smaller fractions of code into logical units *Cliques* and *Compartments* and define directional memory access and control-flow policies among them. So the actual compartments and access rules are entirely made by the programmer and HAKC itself, does not check or optimize isolation policies. The hardware mechanisms used to enforce the derived policies are called *pointer authentication* (PAC), introduced in

ARMv8.3 and *memory tagging extension*(MET), introduced in ARMv8.5-A. The basic principle of TAC is to have pointers cryptographically signed by their legitimate owner, store the signature in the pointer and only allow access based on pointer authentication. MET allows to assign memory regions to different tags and use those groupings to manage access policies. Measured with different microbenchmarks the overhead for HAKC policy enforcement on the ipv6.ko LKM imposed a runtime overhead of 1.6% to 24% for individual benchmarks.

To overcome the difficulty of manually identifying dependencies among components *FlexC*[38] was later presented as an approach to automatically generate the compartmentalization policies that HAKC requires as interface definition among components. The authors argue that in particular in larger system hand-written annotations are error prone and might also lead to inefficient choices. Like HAKC itself, FlexC uses static information from the input code, to generate a data access graph from the input program. To account for indirect data access via pointers the analysis conservatively overestimates potential dependencies. Dynamic data i.e. test runs of the input code can additionally be provided to weighten the data dependencies by frequency of access and size of data. To start the derivation FlexC automatically assigns compilation units (in C usually one ore more files) as smallest entity of isolation. By defining the target number of compartments, the programmer can initialize a greedy fusion of nodes after the initial graph was build, generating a coarser compartmentalization. It is also possible to manipulate the graph using a GUI for FlexC. The output of FlexC is then directly integrated to the compilation process using HAKC.

Since most hardware-related, system-level code is still written in C and C++, the approach of *Spons & Shields*[43] directly builds on these languages. However they focus rather on a scenario commonly encountered in cloud deployments, namely the usage of *Trusted Execution Environments* based on Intel SGX TEEs. TEEs are increasingly popular for secure cloud deployments and while they where initially used to encapsulate only specific critical user applications, they are now used to move whole OSes into a single secure environment e.g. into an Intel SXG enclave. The identified problem is, again that this creates large code spaces without internal access restrictions and possibly including insecure, third party libraries, which contradicts the defense in depth approach. The next problem the authors notice is that compartmentalization is often enforced along processes. But it is hard to redesign an application to encapsulate different concerns and security levels into different processes. Also the trust model of TEEs does not include the host, while process isolation techniques are based on the primitives the host OS and hardware (MMU) provide. The Spons& Shields framework (SSF) described in the paper consists of an API to manage the two core abstractions *Spons*, which encapsulate units of execution, e.g. POSIX processes or libraries and *Shields* which define hierarchical memory access regions. The programmer needs to introduce *Spons* and *Shields* according to her requirements in the code and link against the musl standard C library. During compilation the framework will insert TEE primitives to enable the SSF runtime to enforce memory boundaries between the Spons. To benchmark their technique they used the scenario of a web application consisting of an NGNIX server, a PostreSQL data base (with medical data), an SSL library for crypto and a business logic application in PHP. They compared request latency for a single TEE vs. multi-TEE compartments vs. Shielded deployment and found that shields increase latency by about 1.7 times using Shields while multiple TEEs increased latency about 4.4 times compared to the

single TEE version.

PKRU-Safe[27] is an approach with a slightly different target. Instead of vertical compartmentalization it intends to enforce a separation between memory safe and memory unsafe code in Rust applications. The identified problem is that safe languages in this case Rust are already available and it should be possible to interact with unsafe language components without risking memory corruption. So the whole point is to enforce hierarchic/semi-permeable memory isolation between Rust code and unsafe languages. The concept is to have the developer explicitly define the interface between trusted and untrusted parts of the code. This is done via annotations in the projects build file and the untrusted libraries. With per library annotations the adaptation overhead is significantly smaller than in annotating single functions or data access. To analyse the code and track heap allocations PKRU-Safe uses custom plugins to rustc and LLVM tooling. To identify the usage sites of heap allocated data the input program is run with provided profiling input to dynamically identify when the untrusted component accesses data allocated by the trusted component. The gathered access data are used to categorize data allocation into unsafe and safe memory, in particular data that is used by unsafe code is considered belonging to the unsafe code. At runtime this distinction is used to enforce the policy that unsafe code can never access safe memory. To move such data into the untrusted component entirely and to handle the two memory pools at run time the `liballoc` to augment it by dedicated primitives for unsafe memory allocation. The authors describe this runtime behavior as *compartment-aware* heap allocation. By keeping heap section of safe and unsafe memory objects distinct, PKRU-Safe can implement the policies using MPK protection for the trusted heap section. Notably this policy only applies to data on the heap. Stack data is assumed to be protected by other mechanisms. They apply the concept to Servo, a layout-engine written in Rust and tested the performance overhead with different benchmark suites (Dromaeo, Kraken, Octane and JetStream2). They found the altered allocation mechanism to cause an average runtime overhead of 6% above baseline. The overhead imposed by context switches involving MPK depended on the concrete benchmark with an average of 11% and a maximum of about 30%.

Contrary to the Ohua approach, these techniques are currently bound to a particular language and particular hardware mechanisms. On the one hand this enables a deeper analysis of the code, in particular if also dynamic information is considered. It can also lower the imposed overhead, because a distinction between memory sharing and memory transfer can be made. On the other hand it limits the flexibility. All of the approaches require additional programmer information to identify the target components, but also to shape the intended isolation policies. In how far data of one component is still accessible to the other is in case of Ohua, depending on the chosen architecture integration and the programmers compliance to the programming model, in case the architecture permits the usage of shared references.

5 Evaluation and Lessons learned

In the previous chapter we described how a concrete program can be transformed in such a way that compilation to a distributed program with local states becomes possible. The next question is now whether we can derive an automated, formal process from this, i.e. whether we can implement the necessary transformations in Ohua. It is also important to know what costs in terms of runtime costs and memory, but also in terms of restrictions and effort for the programmer would be associated with the implementation in Ohua. The fact is, even though the smoltcp application now structurally meets the set criteria, we cannot yet compile it with Ohua or run it in the m3 operating system.
Therefore, in this chapter we summarize what we have learned so far about the necessary transformations and what preliminary findings we have about the performance implications and propose further steps in developing the Ohua compiler.

5.1 Learnings from Rewriting

The restructuring of smoltcp was done in several phases and on the basis of the concrete example code. In order to find out which of these phases could be realized in Ohua in the future, we look here again at the steps required for this and the necessary knowledge to be able to take these steps.
We faced several problems along the way. Concretely we needed to

1. encapsulate code in components and define the new interfaces of those components
2. lift stateful component use into scope
3. make states composable, and control flow among them unidirectional
4. adapt the code to the system interface of M^3
5. make state use local, such that no stateful component is ever send among nodes

1. Identifying components and defining their Interaction: The interaction between the `ip_stack` and the `device` where clearly defined. However, for the interaction between the `ip_stack` and the `store` we had to make decisions on which of the processing steps in scope should be assorted to which component. In Section 3.1.2, we discussed several options and their dependency on code structure. In conclusion, identifying components, defining their interaction and encapsulating code accordingly is not a decision that can be made automatically by our compiler. In particular if we only have static information, we cannot predict the size and usage frequency of components at runtime to make sensible decisions on effective communication interfaces. Also the compiler cannot infer what security implications components have. At the moment, the compiler determines what a stateful component is based on the input syntax alone. If the compiler should encapsulate components other than the set of stateful variables used in the compile scope, it will need another source of information, e.g. special annotations given by the

programmer.

2. How can state usage be lifted into Scope? : The basis for the control flow refactorings conducted in this work, was making all stateful calls to components i) visible to the transformations and ii) explicit. We began refactoring by inlining function call to lift `device` calls into scope. Inlining pure functions is already done in Ohua and requires those functions to comply with the programming model. So the first takeaway is that if the compiler shall be able to inline code automatically, the programmer either already needs to know which code this is going to be or has to comply with the programming model in all of the code. This is not trivial, as some aspects of the model cannot be automatically enforced and might be pretty subtle. Inlining stateful functions is currently not done in Ohua, meaning that methods cannot be treated as algorithms. In the transformations we also only inlined e.g. calls to the `socket` as an intermediate step and re-encapsulated parts of the inlined method again. Given that we aim to compile isolated components, the inlining of methods will probably become no topic. To implement the transformations from this work in the compiler it would be necessary instead to be able to transform methods without inlining them into algorithms. The actual lifting can be done, by returning the control flow to the main scope.

When it comes to making state usage explicit, the transformation process can be fairly involved. To transform the implicit usage of `device` via tokens, required an understanding of the reference usage, the effects each part of the processing would have on the components and in conclusion the reference-free separation of processing in the `ip_stack` from processing in the `device`. So this steps could not have been derived from syntactic information. Consequently we must require all state usages to be explicit in the first place.

3. How can we make state use local and composable? : After lifting all usages of the `device` as explicit calls into the `ip_stack.poll`, we essentially just needed to move calls to the `device` into the outer loop and merge them into a common call. The key insight here was that it requires a) a dispatch method `process_call` representing the *only* outer interface of each component b) a serializable representation of the control flow point each call to this method should be dispatched to and c) in case control flow points where originally only parts of methods a mechanism to decompose such method into addressable syntax constructs. For the `device` the definition of `process_call` was straight forward. Due to the previous transformations, the control flow points where exiting method calls with explicit arguments. Those method calls, including their arguments where represented by an **`enum`** and `process_call` would dispatch the call to the concretely method by pattern matching. The return type of `device.process_call` is the corresponding representation **`enum`** on the `ip_stack`. For the `ip_stack` we had to decompose the `poll` method into callable code blocks. This required a form of lambda-lifting of those blocks. Contrary to lambda-lifting in pure functions, we did not turn all enclosed variables into arguments. Instead we made variables that are not send to other components part of the state, to retrieve and set them before and after each block.

Regardless if variables where send or stored between method calls, they needed to be free of internal stack references. So, although the resulting `process_call` will not be compiled to data flow, we will

likely have to extend the programming model assumption to the code to be transformed that way. In particular we will need to require all variables that will be stored to be owned and all variables that will be send to implement the required serialization. A concept that is structurally very similar to the refactorings made here is the transformation into continuation passing style. Therefore, we will deal next with it whether and under which conditions we can develop automatic transformations for Ohua from it.

4. Can we automatically adapt code to different operating systems?: The short answer is no, not in general. System call APIs are syntactically mostly indistinguishable from normal function calls. Ohuas backend integration can adapt the generated DFG to exiting isolation mechanisms, because the compiler itself introduced the abstractions for nodes and channels and can map them to the according architecture. It cannot identify system accesses, like file system access or network access in the input code. A possible remedy was to provide some kind of annotation that can be used by the specif architecture integration. However, as the example of `phy_wait` demonstrated, the concrete a API of a target architecture also influences the general structure of the input code. The most likely solution is that the programmer needs to use the API and available libraries of the target architecture already in the input program. If the target architecture is a distributed system or a different OS, this also fundamentally calls into question the promise of easy testability of the input code. For m3, for example, it is necessary to use the m3 `device` implementations already in the input code, which is why it cannot be tested in a normal Linux environment.

5. Can we make states entirely local?: In the final program structure each state is used exactly once. But this is not enough to create local states in the compiled program. Two problems remain. The first problem is that Ohua currently does not support state threads for branch instructions and recursion. This was intended so far, because both constructs potentially need to send state to nodes, for example to both branches of an if-else statement. The second problem is that the states themselves must be initialized. In a naive translation to a data flow graph, this would mean that there is one node that creates the state and one node that uses the state. For reasons of code efficiency and code size, we do not want to restrict the states to a certain size or to the use of serializable data types.

To solve the second problem we currently see two possible options. Firstly we can identify the initialization node and the usage node for each state and fuse them to a single node in the backend processing. This requires all states to be initialized by independent functions. Another possibility is to implement wrapper types for each state. Those would hold the actual state as an `Option` field, or an equivalent representation in other languages. It would also implement the `process_call` method, mostly dispatching calls to its wrapped state. The `process_call` implementation of the wrapper would however additionally implement a call to initialize the inner state. Evaluating those and potential other solutions will be part of the future work.

Solving the first problem will be one of our next tasks. Currently, Ohua's concept of state threads is extended to branching and recursion. In our example, the states `ip_stack`, `device` ,and `store` are not used after the loop. Thus, an appropriate dependency analysis in the compiler could not only fuse the initialization with the usage of the state, but also prevent the state from being sent to additional nodes

after usage.

5.2 Preliminary Performance Measurement

It is clear that the transformations described here will have a negative impact on memory consumption and performance. Since we cannot compile the application at the moment, we looked at a comparison between the original and the adapted version of the application. We used the "Yahoo! Cloud Serving Benchmark" YCSB [11][1] for this purpose. YCSB can be adapted to benchmark new data base applications by implementing a custom client to connect to the concrete data base and to generate Read,Insert, Update, and Delete requests in an appropriate form.

Normally, this client opens a new connection to the server for each request and closes it after receiving the response. The problem is that, unlike other TCP APIs, smoltcp does not dynamically create handlers for each request. Since the library should be usable without allocation, there are only sockets and packet buffers that the programmer explicitly creates in advance. Therefore, unlike the Rust `TCPStream` API, for example, the server socket can close the previous connection in a loop pass while a new request is already arriving. The result is frequent connection failures and aborted measurements in YCSB. There are several ways to solve this problem. We chose the option to establish a permanent connection between client and server for the measurements. The disadvantage of this is that individual requests are no longer separated at the TCP level, but fragments of successive requests would have to be processed simultaneously by the application. Since the key-value store works the same in both cases, and since we are only interested in the performance of the network stack, we do not parse and reassemble the requests at this time, and respond to all requests with a standard response.

Since YCSB is written in Java and M^3 does not provide a Java runtime and due to time constraints, we ran preliminary measurements on a standard Ubuntu 20.04.1. installation, on an Intel Core i7-8565U machine with 4 cores and 16GiB RAM. YCSB allows to specify the test scenario as so called *workloads*. These workloads define the kind, number and size of request to be generated and sent. As currently only a default answer is returned, there is no performance difference to be expected between write, update and insert operations. A difference is expected for read operations however, because read requests contain significantly less payload that has to be processed by the network stack. So we ran two workloads composed of insert requests only, and read and update and read requests in equal amounts. In either case 1500 operations were executed on 1000 entries, with each entry having three fields of ten bytes length. An excerpt of the results is shown in Table 5.1. Obviously in both variants there is a hug

smoltcp variant	Update			Read			Insert		
	min	99%-Per.	max	min	99%-Per.	max	min	99%-Per.	max
original	23	114	670	6	10^3	10^3	23	386	$9{,}2^3$
adapted	85	10^4	60^6	24	10^4	80^6	28	$8{,}5^4$	89^4

Table 5.1: Minimal, maximal and 99% percentile of response time in micro seconds for Update, Read and Insert requests after preliminary performance measurements in micro seconds

[1] available on `https://github.com/brianfrankcooper/YCSB`

Obviously, there is a considerable spread between the minimum and maximum response times for both variants. Both the 99% percentile values and the maximum values for the adapted variant are extremely high, with response times of up to 80 seconds. It is also noticeable that the response times for Read requests are, as expected, shorter in the optimal case than for Insert and Update, but on average they take significantly longer. The latter is probably a bug in the test setup that causes relatively small Read to be sent and answered only when they can be combined into a larger package. However, this does not explain the extreme deviations of the rewritten variant. The differences in the minimum response times of the two variants seem plausible. It is therefore necessary to investigate further whether this effect is due, for example, to an inappropriate implementation of the wait mechanism.

We should investigate this issue before making further measurements. In addition, YCSB only suitable to a limited extent for our use case, as the application content of packets is less relevant for measuring network stack performance than the number, frequency, and size of incoming and outgoing packets or, for example, the frequency of failed connection attempts. We should therefore look for alternatives to YCSB for further measurements.

5.3 State of Compilability And Future Work

While the code now fulfills all the properties structurally needed, we still cannot compile it with Ohua. One option we have complying with the supported input syntax is to fuse both loops. The result would be, that each loop goes through all three components, which is very inefficient. So we aim to extend Ohuas supported syntax to include stateful computation in branching and recursion statements, before we can finally compile the application. Since a solution to this problem is already foreseeable, we will not discuss possible implementations further here.

However, there are other syntax constructs that Ohua does not currently support, but which would significantly increase usability. Specifically, in the course of this work while loops, match expressions, and expressions for runtime exceptions that needed to be reformatted so that they could be compiled. Therefore, we briefly address here how these expressions might be directly supported in the future. Also, the M^3 backend currently has limitations that we would like to address in the future and briefly discuss here.

While Loops A very common pattern of server applications is to run in an endless loop. As neither loop nor while are supported by the given Ohua implementation. So while thy can be rewritten to recursion by hand, integrating an automatic transformation for them would significantly improve Ohuas usability. The frontend language of Ohua is purely functional and hence does not entail while loops. The functional equivalent of a while loop is a recursive function, with the following case distinction: If the condition of the while loop is true, the inner code will be executed and a recursive call is made, otherwise the function returns. After every run of the original loop, local bindings go out of scope and only variables that lived outside the loop 'survive' to the next iteration. Translated to a recursive function this is equivalent to using all variables from the outer scope as arguments to the recursive call and return them when the recursion ends. This is illustrated in Listing 5.1 which shows a simple while loop in Rust and a functional equivalent using recursion.

```
fn algo() {
  let mut i:i32 = 0;
  let mut state:State = State::new();
  while check(i) {
    let local = use(i);
    state.mutate_with(local);
    i = i + 1
  }
  stateObj
}
```

```
fn algo() {
  let mut i:i32 = 0;
  let mut state:State = State::new();
  (i , state) = rec_while(i, state)
  stateObj
}

fn rec_while(i:i32, state:State)
  -> (i32, State) {
  if check(i){
    let local = use(i);
    state.mutate_with(local);
    i = i + 1;
    rec_while(i, state)
  } else {
    (i, state)
  }
}
```

Listing 5.1: To express a While-Loop in the purely functional Ohua frontend language, we can rewrite it to a recursive function

We want that transformation to be valid/suitable for all language integrations and keep the creation of new language integrations simple. Therefore we should augment Ohuas frontend language with a Whileconditionbody expression, where condition and body are the respective components of the original while-loop transformed to expressions of the frontend language. The return type of the recursion should be determined by a bottom up analysis to only return values that are used after the recursion. The simple transformation shown in Listing 5.1 does not account for Ohuas current limitation in recursion syntax, i.e. that branches must only contain the return statement. So the simpler translation of while loops is another argument to fix this restriction, especially since it is not otherwise based on content.

Match Expressions Ohuas frontend language is functional and functional languages are notorious for pattern matching. However it does not entail a syntax representation for pattern matching as used in Rusts `match` expressions. We cannot trivially translate a `match` expression into nested binary branching statements, because the arms of match are guarded by pattern matches and not by boolean conditions. However, if we wrap the actual pattern matching out of the compile scope, we can transform the guards either to boolean values. The principle is shown in Listing 5.2

```
// matching in a function
let x: type = match some_call() {
    pattern1 => /*closure 1*/
    pattern2 => /*closure 2*/
    _ => /*closure 3*/
}
```

(a) Example for matching in pseudocode

```
// replaced function code
let y: type2 = some_call();
let x: type =
   if first(y) {
       /* closure 1 */
   } else if second(y) {
       /* closure 2 */
   } else {/*closure 3*/}

//----------------
// In a new separate library
fn first(y:type) -> bool {
   match y {
       pattern1 => true
       _ => false
  }
}
fn second(y:type) -> bool {
   match y {
       pattern2 => true
       _ => false
  }
}
//  ...
```

(b) Pseudocode for compilation result

Listing 5.2: Match moved to a library function

A more efficient implementation of this idea, is to extend the 'switching range' of control nodes from a binary decision to multiple branches using integer flags. This also happens basically when the Rust compiler lowers either if-else or matches[2]. With this approach the `match` expression can be encapsulated into a single external function, returning the integer index of the matching branch. In the main scope we would still replace the original match function with a nested if-else, checking equality of `y` with the possible indices.

```
fn match(y:type) -> bool {
   match y {
       pattern1 => 0,
       pattern2 => 1,
       _ => false
  }
}
```

Listing 5.3: Rewrite pattern matches to enumerated cases

[2]see https://rustc-dev-guide.rust-lang.org/mir/construction.html#lowering-expressions-into-the-desired-mirrustc docu

Panic – Runtime Exceptions There are different ways of producing and handling runtime errors in Rust. If runtime errors are implemented using the `Result` type and handling of the error by an alternative control flow, it is basically opaque to the compiler i.e. treated like any other type. Apart from that there are basically two types of errors, that we currently cannot handle.
This first one are runtime exceptions, called `Panic` in Rust parlance. They immediately terminate the current thread. This behavior is accessible via is implemented as the `panic!`, but will also be introduced via using functions that can panic. So basically we cannot detect panicking code, based on the syntax unless we compile all code involved. After compilation, only the node executing the panicking code will actually stop running, in a distributed scenario the other nodes will continue running, but data flow will be interrupted at the failing node. It is desirable and necessary to have a mechanism that propagates and handles errors across the distributed program. However, there are several reasons not to attempt this initially through further compiler transformations. These are

1. To catch a runtime exception and propagate the error out of the failed process there are constructs like try-catch blocks. However, this is not the case in all languages and is not provided for in Rust, for example.

2. As already described, it is not possible to syntactically recognize function calls that produce runtime exceptions.

3. Under certain circumstances, various nodes should not simply be terminated, but for example resource should be returned beforehand or communications with external processes should be gracefully terminated.

Therefor we argue that runtime exceptions should not be treated by the compiler. Instead the programmer should implement proper error propagation and handling.
The second type are early returning errors, implemented in Rust via the `?` symbol (formerly via the `try!` macro). It is syntactic sugar for unwrapping either a `Result` or an `Option`. If successfully the unwrapped value will be processed further, if an `Error` or a `None` was encountered, they are early returned from the function. For an expression `expr ?`, whether `expr` returns a `Result` or an `Option` we could transform the code by i) removing the `?` and bind the result of `expr` to a variable if that is not already done and ii) wrap the downstream execution in a branching statement such that in case of successfully unwrapping the rest of the function is executed, otherwise the result is returned. A small example is shown in Listing 5.4 below. Obviously we will need to handle early returns for this solution.

```
fn example() -> Result<T,E> {
    let x : type = might_error()?;
    // rest of the function
}
```

```
fn example() -> Result<T, E> {
    let x:Result<type> = might_error();
    if x.is_ok() {
        // rest of the function
    } else {
        return x
    }
}
```

Listing 5.4: Refactoring option for `?`, early returning errors

The M^3 Backend

We currently run all nodes as separate processes on one tile. To leverage the full potential of M^3, we may change the backend implementation to actually allocate separate tiles for each process. As M^3 is designed to integrate different hardware components as tiles, this will be an interesting step towards supporting heterogeneous platforms.

A limitation concerning the supported programming model is, that so far we did not implement the handling of environment variables, i.e. data initialized outside the algorithms and passed as arguments or just used in a shared scope. For the present example application, this was not an issue. We were compiling a `main` function, which does not take input arguments. However we might want to also compile for libraries to M^3, where translated algorithms may need external input. How this can be implemented, depends on how the processes are created, i.e. on a single or multiple tiles and on whether the environment variable was defined inside the compile scope e.g. global constants or is actually passed in from processes calling the compiled code. In this context we will also need to evaluate whether its possible and necessary to generate the `xml` configuration files M^3 uses to declare interfaces and names of applications.

Finally a problem that we will not be able to handle automatically is the translation of system service accesses, such are file system access, or the usage of device drivers. We saw for example, that we needed to adapt the implementation of `phy_wait` based on M^3 implementations for network devices and process waiting. As described in Section 4.1 FlexOS and CubicleOS handle this problem by providing the programmer a) with a fixed set of supported libraries they can use and b) annotations for system calls, such as memory allocation the compiler can use to automatically replace system call appropriately for the target system. So likely for us adaptation to operating system features, other than the generation and direct communication of processes, will also rely on information from the programmer and the usage of suitable interfaces already in the input code.

6 Conclusion

Isolating components of a kernel or larger programs from each other makes sense both for security reasons and for better scalability. But adapting existing code bases is time-consuming and complicates development and testing. This is also true, for example, when programs written for monolithic or unikernels have to be rewritten for microkernel-based systems like M^3 with isolated drivers and system components. The Ohua compiler tries to solve this problem by automatically identifying independent sections of a program and making them separate threads or processes, depending on the architecture chosen.

In this work we have looked at whether it is possible to compile programs with Ohua whose structure does not yet contain the desired independent components. Using the concrete example of a server application, we looked at which transformations are necessary to create a program in which the TCP/IP stack, the network interface and the user application are independent, stateful components. By means of a backend integration for the microkernel-based operating system M^3 , a program transformed in this way could be compiled directly from running on a uni- or monolithic kernel to running on a microkernel based OS.

To achieve this, the components must only be used once in the compiled program. The communication between them must be realized via the arguments of a single function call, and the control flow of the entire program must be adapted accordingly. As it turned out, part of this problem can be formally described by transformation to Continuation Passing Style and Defunctionalization, and could be further developed to transformations in Ohua. Other necessary steps are not possible without additional demands on the programmer. This includes, among other things, the identification of the target components if this is not apparent from the initial code structure. In general, lifting code into scope that was encapsulated in function or method calls is problematic because it is the programmer's responsibility to respect the constraints of the programming model, e.g. that function arguments need to be serializable for the selected backend. As the transformations that Ohua performs on the code become more complex, it may become more difficult for the programmer to know in which areas of the code to adhere to the programming model.

Comparing Ohua with other approaches for compartmentalization we have seen that Ohua is more flexible since it is not directly build upon a single input language, compiler toolchain or particular hardware mechanism. However, as we have seen in the adaption to M^3 , we face the same problems when it comes to adaptations between different operating systems, namely that system calls can not be automatically identified from the syntax and replaced by Ohua. It might be worthwhile to consider adding annotations to Ohuas API in future versions, to enable e.g. the automatic encapsulation of components or the use of different data transfer mechanisms. Otherwise the input program already needs to use appropriate system calls and components, which contradicts the original idea of being able to develop the code inde-

pendently of the target architecture.

For further work, we therefore need to clearly distinguish which transformations we can and want to implement in Ohua, and which steps should be left to the programmer.

Bibliography

[1] National vulnerability database.

[2] Redox project website. `https://www.redox-os.org/`. Accessed: 2022-07-16.

[3] The rust programming language. `https://doc.rust-lang.org/book/`. Accessed: 2022-12-20.

[4] The rust reference. `https://doc.rust-lang.org/reference/`. Accessed: 2022-12-20.

[5] smoltcp. `https://m-labs.hk/software/smoltcp/`. Accessed: 2022-05-31.

[6] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. 1986.

[7] Atul Adya, Jon Howell, Marvin Theimer, William J Bolosky, and John R Douceur. Cooperative task management without manual stack management. In *USENIX Annual Technical Conference, General Track*, pages 289–302, 2002.

[8] Nils Asmussen, Sebastian Haas, Carsten Weinhold, Till Miemietz, and Michael Roitzsch. Efficient and scalable core multiplexing with m^3v. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 452–466, Lausanne, Switzerland, February 2022. ACM.

[9] Nils Asmussen, Michael Roitzsch, and Hermann HÃ¤rtig. M3x: Autonomous accelerators via context-enabled fast-path communication. In *USENIX Annual Technical Conference (ATC)*, Renton, WA, USA, July 2019. USENIX.

[10] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E Engelstad, and Kyrre Begnum. Includeos: A minimal, resource efficient unikernel for cloud services. In *2015 IEEE 7th international conference on cloud computing technology and science (cloudcom)*, pages 250–257. IEEE, 2015.

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[12] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

[13] Olivier Danvy. Defunctionalized interpreters for programming languages. *ACM Sigplan Notices*, 43(9):131–142, 2008.

[14] Olivier Danvy and Kevin Millikin. Refunctionalization at work. *Science of Computer Programming*, 74(8):534–549, 2009. Special Issue on Mathematics of Program Construction (MPC 2006).

[15] Sebastian Ertel, Justus Adam, and Jeronimo Castrillon. Supporting fine-grained dataflow parallelism in big data systems. In *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 41–50, 2018.

[16] Sebastian Ertel, Justus Adam, Norman A Rink, Andrés Goens, and Jeronimo Castrillon. Stclang: state thread composition as a foundation for monadic dataflow parallelism. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*, pages 146–161, 2019.

[17] Sebastian Ertel, Christof Fetzer, and Pascal Felber. Ohua: Implicit dataflow programming for concurrent systems. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, pages 51–64. 2015.

[18] Sebastian Ertel, Andrés Goens, Justus Adam, and Jeronimo Castrillon. Compiling for concise code and efficient i/o. In *Proceedings of the 27th International Conference on Compiler Construction*, pages 104–115, 2018.

[19] Paul Graunke, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Automatically restructuring programs for the web. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 211–222. IEEE, 2001.

[20] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. {CertiKOS}: An extensible architecture for building certified concurrent {OS} kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, 2016.

[21] Per Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, 1970.

[22] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[23] Dan Hildebrand. An architectural overview of qnx. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126, 1992.

[24] Galen Hunt, James R Larus, Martın Abadi, Mark Aiken, Paul Barham, Manuel Fahndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, et al. An overview of the singularity project. 2005.

[25] Intel Intel. and ia-32 architectures software developer's manual. *Volume 3A: System Programming Guide, Part*, 1(64):64, 64.

[26] Liedke Jochen. On μ-kernel construction. In *Proceeding of 15th ACM Symposium on Operating System Principles,(SOSP' 95)*, pages 237–250, 1995.

[27] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. Pkru-safe: automatically locking down the heap between safe and unsafe languages. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 132–148, 2022.

[28] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an os microkernel. *ACM Trans. Comput. Syst.*, 32(1), feb 2014.

[29] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, et al. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 376–394, 2021.

[30] Chun Kit Lam, Stephan Maka, David Nadlinger, Chris Ballance, and Sébastien Bourdeauducq. Combining processing throughput, low latency and timing accuracy in experiment control. *arXiv preprint arXiv:2111.15290*, 2021.

[31] John Launchbury and Simon L Peyton Jones. Lazy functional state threads. *ACM SIGPLAN Notices*, 29(6):24–35, 1994.

[32] Hugo Lefeuvre. Flexos: easy specialization of os safety properties. In *Proceedings of the 22nd International Middleware Conference: Doctoral Symposium*, pages 29–32, 2021.

[33] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[34] Jochen Liedtke. A persistent system in real use-experiences of the first 13 years. In *Proceedings Third International Workshop on Object Orientation in Operating Systems*, pages 2–11. IEEE, 1993.

[35] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *ACM SIGARCH Computer Architecture News*, 41(1):461–472, 2013.

[36] Anil Madhavapeddy and David J Scott. Unikernels: the rise of the virtual library operating system. *Communications of the ACM*, 57(1):61–69, 2014.

[37] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. Preventing kernel hacks with hakc. In *Proceedings 2022 Network and Distributed System Security Symposium. NDSS*, volume 22, pages 1–17, 2022.

[38] Derrick P Mckee. *Novel System Compartmentalization and Reverse Engineering Methods*. PhD thesis, Purdue University Graduate School, 2022.

[39] Ruslan Nikolaev and Godmar Back. Virtuos: An operating system with kernel virtualization. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 116–132, 2013.

[40] Sandro Pinto and Nuno Santos. Demystifying arm trustzone: A comprehensive survey. *ACM computing surveys (CSUR)*, 51(6):1–36, 2019.

[41] John C Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*, pages 717–740, 1972.

[42] John M Rushby. Design and verification of secure systems. *ACM SIGOPS Operating Systems Review*, 15(5):12–21, 1981.

[43] Vasily A. Sartakov, Daniel O'Keeffe, David Eyers, Lluís Vilanova, and Peter Pietzuch. Spons & shields: Practical isolation for trusted execution. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2021, pages 186 – 200, New York, NY, USA, 2021. Association for Computing Machinery.

[44] Vasily A Sartakov, Lluís Vilanova, and Peter Pietzuch. Cubicleos: a library os with software componentisation for practical isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 546–558, 2021.

[45] Lin Tan, Ellick M Chan, Reza Farivar, Nevedita Mallick, Jeffrey C Carlyle, Francis M David, and Roy H Campbell. ikernel: Isolating buggy and malicious device drivers using hardware virtualization support. In *Third IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC 2007)*, pages 134–144. IEEE, 2007.

[46] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E Porter. Cooperation and security isolation of library oses for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[47] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, 1992.

[48] Andrew Whitaker, Marianne Shaw, and Steven D Gribble. Scale and performance in the denali isolation kernel. In *5th Symposium on Operating Systems Design and Implementation (OSDI 02)*, 2002.

List of Figures

List of Listings

List of Tables

A Appendix

A.1 Rust Language Integration

Block of Statements:			
$block$	::=	$\{s;\ \ldots;\ s\}$	statements
Statements:			
s	::=	e	statement returning a value
	|	e**;**	statement not returning a value
	|	**let** pat **=** e	local definition
Expressions:			
e	::=	x	variable bindings
	|	$\mathbf{1}, \mathbf{2}, \mathbf{3}, \ldots$ | **true** | **false**	literals
	|	e **(**$e, \ldots, e$**)**	function calls
	|	$e\ callRef$ **(**$e, \ldots, e$**)**	method calls
	|	**(**$e,\ \ldots,\ e$**)**	tuples
	|	$e\ +\ e \mid e\ -\ e \mid e\ >\ e \mid e\ ==\ e \mid \ldots$	binary operations
	|	$-e \mid !e \mid *\, e$	unary operations
	|	**if** $e\ block$ **else** e	conditional with optional else branch
	|	**for** pat **in** $e\ block$	
	|	**move |** $arg,\ \ldots,\ arg$ **|** e	closure
	|	$block$	block expression, i.e. block returning a value
Call References:			
$callRef$	::=	z**.**y**.**$x\ [gernericArg]$	namespaced binding
Patterns:			
pat	::=	$x \mid (x,\ \ldots,\ x) \mid \_$	bindings, tuples or wild cards

Table A.1: Subset of Rust grammar, that is accepted by the Rust integration frontend

A.2 Additional Source Code Excerpts

```
pub fn poll<'a, D>(
    mut ip_stack: Interface<'a>,
    timestamp: Instant,
    mut device: D,
    mut sockets: SocketSet<'static>)
    -> ( Result<bool>, Interface<'a>, D, SocketSet<'a>)
    where D: for<'d> Device<'d>
{

    ip_stack.set_inner_now(timestamp);
    // .. fragment handling

    let mut readiness_has_changed = false;

    loop {
        let processed_any = ip_stack.socket_ingress(device, sockets);
        // Begin of inlined ip_stack.socket_egress()

        let mut emitted_any = false;

        for item  in sockets.items_mut() {
            if ip_stack.item_meta_egress_permitted(item)
            {
                let mut neighbor_addr = None;
                let result:Result<()>;
                let packet_or_ok =
                    // socket function wrapped to happen inside the ip_stack
                    // component
                    ip_stack.match_socket_dispatch_before(item);
                if is_packet(&packet_or_ok) {
                    let (response, response_and_keepalive) =
                        from_packet(packet_or_ok);
                    neighbor_addr = as_optn_addr(&response);
                    let sending_token = device.transmit();
                    if sending_token.is_some() {
                        let local_dispatch_result = ip_stack.inner_dispatch_local(response, None);
                      if let Ok((packet, timest)) = local_dispatch_result{
                            let send_result =
                                device.consume_token(timest, packet, sending_token.unwrap());
                            if send_result.is_ok() {
                                // socket function wrapped to happen inside // the ip_stack component
                                result =
                                ip_stack.match_socket_dispatch_after(item, response_and_keepalive);
                                emitted_any = true;
                            } else {
                                result = send_result
                            }
                        } else {
                            result = Err(local_dispatch_result.unwrap_err());
                        }
                    } else {
                        net_debug!("failed to transmit IP: {}", Error::Exhausted);
                        result = Err(Error::Exhausted);
                    }
                } else {
                    result = Ok(());
                }

                let maybe_break = ip_stack.handle_result(result, item, neighbor_addr);
                if maybe_break {
                    break
                    }
                }
            }
            // End of inlined ip_stack.socket_egress()

        if processed_any || emitted_any {
            readiness_has_changed = true;
        } else {
            break;
        }
    }
    (Ok(readiness_has_changed), ip_stack, device, sockets)
}
```

Figure A.1: After inlining all functions using the `device` and re-encapsulating all code that concerns only the `ip_stack` we get an interleaved usage of `ip_stack` and `device` in the sending loop. Due to multiple mutable borrows of both components in the loop, this code will not compile.

Acknowledgments

Of course, I would like to thank the people who helped me with this thesis. To my supervisor Sebastian Ertel for his time, help and brainstorming together, as well as Felix Suchert for helping me master Rust problems and Nils Asmussen for helping me understand M^3 .

But I wouldn't have written this thesis, and probably not a single line of code, without Vera Zeidler and Sebastian Flügge. Without Sebastian I would not have had the idea to study computer science, and that would have been a pity, because the idea was really good. Without Vera I literally wouldn't be here. Thank you for all your efforts, love and cleverness, I hope you are also a little proud.