Robust and Immediate Resource Reclamation with M³

Viktor Reusch
viktor.reusch@barkhauseninstitut.org
Barkhausen Institut
Dresden, Germany

Nils Asmussen
nils.asmussen@barkhauseninstitut.org
Barkhausen Institut
Dresden, Germany

Michael Roitzsch
michael.roitzsch@barkhauseninstitut.org
Barkhausen Institut
Dresden, Germany

Abstract

Asynchronous programming is a very helpful programming methodology for systems with many threads and long-running I/O. However, care must be taken if systems, e.g., kernels, work with objects that can be dynamically created, used, and destroyed by asynchronous tasks. The typical approach keeps an object alive until all tasks are done with it. However, this can lead to long delays until resources can be reclaimed and performs useless work on already destroyed objects.

This paper proposes a new set of abstractions for object references that enables an immediate resource reclamation. We build upon the Rust programming language and use a combination of static analysis and runtime checks to guarantee that suspended tasks do not keep objects alive. We additionally leverage language features to ensure that destroyed objects cannot be accessed anymore, making it robust. We perform a case study with the M³ kernel, which has many asynchronous system calls and manages shared kernel objects. In the evaluation, we show that our approach incurs a modest performance overhead even for system-call heavy workloads.

ACM Reference Format:
Viktor Reusch, Nils Asmussen, and Michael Roitzsch. 2024. Robust and Immediate Resource Reclamation with M³. In *2nd Workshop on Kernel Isolation, Safety and Verification (KISV '24), November 4–6, 2024, Austin, TX, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3698576.3698763

1 Introduction

Asynchronous programming is a popular technique to hide I/O latency and to increase the number of requests processed concurrently, thus improving throughput. For that reason, we observe its adoption in modern programming languages [7], frameworks [18], software [10], and operating system kernels [1]. On a fundamental level, "asynchronous" means that an executed task is suspended after starting an I/O request and resumed only after the I/O operation is finished. In the meantime, the underlying thread can continue to execute other tasks until they invoke an asynchronous operation as well. Thus, multiple tasks can concurrently make progress while others are blocked.

While increasing throughput, this programming methodology also introduces additional complexity. For example, working on shared objects across asynchronous invocations is not trivial if objects are created and destroyed dynamically. Due to the concurrent nature of asynchronous programming, object references that were retrieved before awaiting an asynchronous call may be invalid once the program flow continues. Other tasks could have destroyed the referenced objects in the meantime. To avoid dereferencing dangling pointers and use-after-free, one might employ reference counting or garbage collection. However, these techniques have multiple downsides. First, while tasks wait for I/O completion, they also delay the destruction of objects and thus the reclamation of associated resources. This problem is exacerbated when tasks wait for responses from untrusted communication partners for potentially arbitrarily long periods of time. As a consequence, there can be a large gap between the point of the intended deletion of an object and the actual reclamation of the resources. Second, tasks might perform useless work on objects that have already been logically deleted while they were suspended. These objects are deallocated at the end of the current function anyway. In the big picture, this implies that systems require more memory to accommodate the lingering objects or else allocations have to wait for memory resources to be reclaimed.

In contrast, *immediate resource reclamation* enables systems to operate with constrained memory resources while keeping high performance. We propose using a reliable way to detect that objects are gone instead of keeping them alive while referenced. This makes it possible to free object resources immediately. However, this methodology also introduces a new set of possible programming mistakes. Thus, our proposed abstractions are designed to be *robust* by employing language, compile-time, and runtime checks that prevent common programming mistakes.

We explore this idea of immediate resource reclamation and detection of gone objects in a case study with M³ [2]. M³ is a hardware/operating-system co-design for heterogeneous many-core systems. It is specifically designed to support untrusted cores and accelerators. M³ runs a single microkernel on a dedicated, trusted core. The other (potentially untrusted) cores are managed by per-core multiplexers. The single M³ kernel handles all system calls but needs to send instructions to the per-core multiplexers to change core-local state like page tables. These instructions are transmitted using asynchronous message passing based on cooperative threads. Like

KISV '24, November 4–6, 2024, Austin, TX, USA

ACM ISBN 979-8-4007-1301-9/24/11
https://doi.org/10.1145/3698576.3698763

other microkernels, the M³ kernel keeps state in form of kernel objects and provides applications access to them via capabilities. This combination of asynchronous programming and dynamic object management makes the M³ kernel a good fit for our case study on resource reclamation.

This paper contributes the design of a set of abstractions for object references that is robust, preventing programming mistakes, and allows for immediate reclamation of object resources. We perform a case study by adding an implementation of the proposed abstraction to the asynchronous M³ kernel. Last but not least, we contribute a performance evaluation of our approach.

2 Design

We design smart abstractions for object references that work in the context of asynchronous computation. The design follows three goals. First, the design must be *safe*. It should not be possible to provoke use-after-free or other memory safety bugs. Second, we require *immediate reclamation* of object resources, i.e., references in other tasks need be become invalid once an object is destroyed. Last but not least, the design has to be *robust*. The two guarantees above must not break due to simple programming mistakes.

2.1 Existing Guarantees From Rust

The M³ microkernel is written in Rust [14]. Rust is a memory-safe programming language without any runtime or garbage collector. Thus, Rust is good fit for writing the M³ microkernel in terms of safety, reliability, and speed.

Important safety properties of Rust are enforced by the borrow checker [11]. This compile-time analysis tool tracks ownership of values across variables, the state and kind of references to these values, and when and through which references values are accessed. Most importantly for us, Rust forbids destroying objects while still holding references derived from them. This guarantees that references do not become dangling and that use-after-free cannot happen. In the context of our work, the borrow checker prevents kernel code from releasing reference counters or discarding object guards while still holding on to references to (parts of) kernel objects. Thus, reference counting and other runtime checks on object references can be constructed in a robust way assuring safety guarantees.

2.2 Runtime Checks

Rust does not enforce all guarantees needed to implement our approach of immediate resource reclamation. Most importantly, we require that kernel objects can be released at any time. This requires that suspended tasks cannot hold object references. However, there is no compile-time guarantee that could be used to assure that object references are not held across asynchronous calls.

Thus, we introduce runtime checks that track the objects queried from the internal object collections. If a reference to such an object is still live when attempting a task switch, an error message is printed and the kernel is aborted. This assures that tasks do not rely on object references being valid across asynchronous calls and guarantees that objects can be freed at the intended moment to reclaim resources.

2.3 Static Analysis

In addition to the guarantees provided by Rust and runtime checks, static analysis can be used to make further assurances about the source code. These checks run at compile time and do not influence runtime performance. We want to clearly highlight which code paths might lead to asynchronous calls and therefore require particular attention. The highlighting is particularly useful as developers have to make sure that no object references are held across asynchronous calls to avoid system aborts. Knowing when these calls might happen is therefore essential. We have defined precise rules regarding which functions are allowed to perform asynchronous calls:

- Every function that directly suspends the current task and allows other tasks to be scheduled must be suffixed with `_async` (subsequently referred to as ASYNC). The `_async` suffix is a clear indication of possible concurrency to the programmer. We have decided against introducing a new keyword or special syntax for ASYNC functions. These approaches would be incompatible with existing Rust tooling.
- Transitively, every function that invokes an ASYNC function must also be suffixed with `_async`.
- All other functions, which do not perform asynchronous operations, must not be suffixed.
- We also need to prevent ASYNC functions from being used under a different name or without a name. Hence, we do not allow pointers to ASYNC function to be passed around or stored and do not allow ASYNC functions to be used in closures.

Note that the static analysis is not strictly necessary if Rust's `async`/`await` feature is used. However, it is required if asynchronicity is for example provided based on green threads as in our case study (see Section 3.1.3 for details).

2.4 Custom Reference Abstractions

We introduce three abstractions for object references based on traditional reference counting but specifically tailored to work with objects across asynchronous calls:

`StrongRc` The `StrongRc` effectively behaves like a reference in traditional reference counting. While at least one "strong" reference to an object is held, the object is kept alive. This reference type is used inside the kernel's object collections. Naturally, these references are allowed to exist across asynchronous calls, which is not a violation of our approach as tasks cannot hold `StrongRc` references directly.

`TempRc` Whenever a task wants to access a kernel object, it retrieves a reference from a collection. The retrieved references will be of type `TempRc`, not `StrongRc`. This kind of reference is not allowed to be held across asynchronous

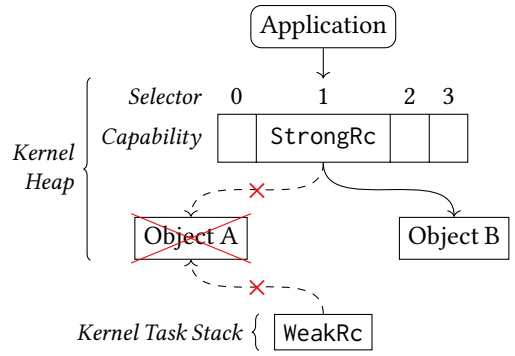

Figure 1. Applications access kernel objects through selectors and capabilities. Selectors can be reused with new capabilities pointing to different objects. Though, weak references can only ever get invalidated.

calls, which is enforced via runtime checks. Thus, concurrent tasks can destroy objects knowing that no other task can still hold a `TempRc`.

`WeakRc` The third reference type is `WeakRc`, which behaves like a weak reference in ordinary reference counting and can be held across asynchronous calls. "Weak" references are created by downgrading `TempRc`s. They must be upgraded again after asynchronous calls to access the object. The upgrade only succeeds if the object is still alive.

One could assume that the `WeakRc` abstraction is unnecessary as the kernel could simply query for the same `TempRc` again after an asynchronous call. However, a new object could use the same selector after the original object occupying this slot had been destroyed. Thus, a query using the same selector might yield a reference to a different, unexpected object. A visualization is given in Figure 1 using the data structures of the M³ kernel as an example. Weak references avoid this problem elegantly by always staying invalid after the object has been destroyed.

2.5 Invalidation of Weak References

The abstraction of weak references also comes in handy when objects are being destructed. Some objects might perform asynchronous I/O during their destruction. While the destructing task is suspended, other tasks could observe and even manipulate the object under destruction, which can easily lead to bugs. Our approach solves this problem by assuring that objects are not reachable once destruction starts, i.e., weak references become invalid. Our abstraction of `WeakRc` makes it possible to forcefully invalidate all weak references even if a strong reference still exists. The forceful invalidation assures that objects are not modified during destruction and that other tasks that still reference such objects abort early.

2.6 Object Mutations

Similar to object destruction, some use cases might require that tasks abort when referenced objects have been mutated. For example, kernel tasks might want to abort a system call when a referenced application changed its state from *running* to *stopped*. To prevent tasks from continuing to work with mutated objects, we could introduce another abstraction, `StableRc`. It behaves like a `WeakRc` but is invalidated not only on object destruction but also whenever the object is modified. Modifications can be detected by combining `StableRc` with, for example, `RefCell` [13]. However, the `StableRc` abstraction might not be useful in all scenarios. For instance, we had no use for this particular abstraction in M³.

3 Case Study with M³

We evaluate the practicality of our proposed design by modifying the M³ kernel. This kernel makes heavy use of asynchronous programming and resource reclamation. It is thus a prime target for our study.

3.1 Background on M³

M³ is a hardware/operating-system co-design developed for heterogeneous systems. It specifically supports untrusted cores and accelerators in a tiled hardware architecture.

3.1.1 Basic Architecture. The processing cores and accelerators in M³ are separated into individual tiles. Each of them is connected to a data transfer unit (DTU), which interposes the communication and the memory accesses of each tile. The DTU offers a unified memory and inter-process communication (IPC) interface, which is especially helpful on heterogeneous systems like M³. Furthermore, it enforces security policies by controlling which memory a tile can access and with whom it can communicate.

3.1.2 Split in Kernel and TileMux. The DTUs have to be programmed by a trusted entity in the system to uphold the promised security. This instance is the M³ kernel, which runs on a separate, single-core tile and is the only entity privileged to program the DTUs.

The M³ system also supports sharing tiles with multiple applications. Every CPU tile runs a local multiplexer called *Tilemux*. The multiplexer preempts applications running on the local tile and switches between application threads following a scheduling policy. Nevertheless, all management is still performed by the single M³ kernel, which has the full control of the system. The kernel regularly needs to communicate with the Tilemux instances to start applications, change scheduling parameters, and alter page tables. Thus, asynchronous calls are an important building block of the M³ kernel, making it a good candidate for our case study.

3.1.3 Green Threads vs. Async/Await. Kernel tasks regularly wait for some event during system-call handling. They

might wait for answers from Tilemux or for semaphore counters to be incremented. While kernel tasks are suspended, M³ uses the available compute resources to handle system calls of other processes. It employs stack-based threads on top of a single hardware thread to achieve CPU multiplexing. This setup is very similar to the well-known green threads of many application frameworks. Thus, we also call the M³ kernel threads "green threads". Each green thread has its own stack allocated from kernel memory, which is also used to store the register state of threads. Switching tasks is as simple as pushing the registers of the current thread to the stack, loading the stack pointer of the other green thread, and popping its registers.

This approach is notably different from the *async/await* feature of Rust [9]. In Rust, async/await transforms code appearing synchronous into a state machine. This state machine allows to execute the asynchronous functions step-by-step, always yielding when the code needs to wait for some event. The constructed state machine also internalizes local variables, which makes a separate stack unnecessary. However, the state machine and all the runtime handling turns out to be more heavy weight and yields worse performance than the green threads in case of the M³ kernel. Nonetheless, we assume that our design for resource reclamation in asynchronous code would apply to a state-machine based implementation.

3.1.4 Capabilities and Kernel Objects. The M³ kernel provides applications with a mechanism for configuring the DTUs. These configurations of the DTU are abstracted as kernel objects like send, receive, or memory gates for IPC and memory access. As typical for microkernels, access to resources and kernel objects is guarded by a capability system. Each application has its own set of capabilities, which are stored inside the kernel and point to kernel objects. Applications can neither manipulate nor forge capabilities directly. Instead, applications only indirectly refer to capabilities by a *selector*, which indexes into their associated capability space in the kernel. This indirection is depicted in Figure 1. The permissions that an application possesses are dictated by the capabilities in its associated capability space. Furthermore, applications can delegate capabilities to other applications to grant them access to the associated resource too.

As many other capability systems, M³ also supports revocation of delegated capabilities. Revocation denies access to resources and reclaims kernel memory quota. The revocation operation can also lead to kernel objects being destroyed when the last capability pointing to them is gone. The combination of revocation and the asynchronous nature of M³ system calls implies that kernel objects can vanish while tasks are suspended. To avoid this, M³, so far, kept strong reference in kernel tasks during system calls. However, keeping strong references has major disadvantages, as mentioned in the introduction. For example, the previous version had a bug that lead to a configured communication channel even if the object it was based upon had been destroyed during an ASYNC call. The modified M³ prevents this bug by design, because references need to be downgraded before ASYNC calls and upgraded afterwards, which detects destructions reliably.

Listing 1. Simplified example of creating and using a semaphore (capability) in the M³ kernel.

```
let sem: StrongRc<Semaphore> = Semaphore::new();
current_activity.insert_cap(selector,
  Capability::new(sem));
let sem: TempRc<Semaphore> =
  current_activity.get_cap(selector).object();
let sem: WeakRc<Semaphore> = sem.downgrade();
some_function_async();
let sem: TempRc<Semaphore> = sem.upgrade()?;
```

3.2 Implementation

To implement our design, we extended the M³ project both at compile and at run time.

3.2.1 Static Analysis. To perform static analysis, we make use of the *Dylint* linting tool [8]. This tools allows to write custom Rust lints similar to the standard *Clippy* [12] lints. Lints can make use of the *rustc* [17]-internal data structures, i.e., the same structures the Rust compiler uses, to find problematic source code and suggest fixes.

We created three custom lints for our static analysis:

1. The first lint asserts that ASYNC functions are only used inside of functions with the suffix `_async`. Our linter therefore visits every regular function in the rustc-internal abstract syntax tree and checks the suffix of every called function. This lint also applies to closures, checking that they contain no ASYNC invocations too.
2. In a similar way, another lint checks that ASYNC functions actually contain at least one invocation of an ASYNC function.
3. Last but not least, a third lint checks that path expressions with names of ASYNC functions are only ever used directly as the called side of a function call expression. This effectively prohibits any storing, moving, or passing around of pointers to ASYNC functions as asked for in Section 2.3.

3.2.2 Abstractions. For our case study with M³ we implement the three abstractions proposed in Section 2.4, `StrongRc`, `TempRc`, and `WeakRc`. Listing 1 presents an example on how these abstractions are used in the M³ kernel.

We have opted for writing our own reference counting instead of relying on Rust's standard library [15]. This way, we can manually invalidate weak references and have more control over the memory footprint of destroyed objects. The weak references of the standard library [16] directly point to the allocated structure on the heap, which contains the object itself, the strong, and the weak reference counters. Thus, one can only reclaim the memory allocated for a kernel object when all strong and weak references are gone. However, for

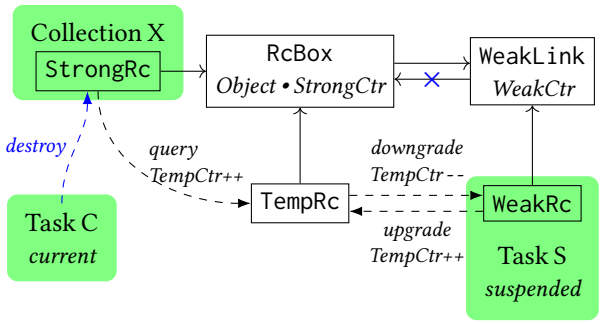

Figure 2. Internal data structures of the proposed design. A reference to a kernel object is queried from a collection, yielding a `TempRc`. A global counter is incremented to track active references. The `TempRc` can be downgraded to a `WeakRc`, decrementing the counter. A `WeakRc` is only an indirect reference to the object and can be invalidated when the object is destroyed.

the M³ kernel we want to be able to free the kernel memory associated with an object as soon as possible. Applications on M³ pay for kernel memory that is allocated for them using their associated kernel memory quota. Hence, the M³ kernel should give back quota spent on kernel objects as soon as possible.

Our implementation allocates a small indirection object, a `WeakLink`, for every kernel object. `WeakRcs` point to `WeakLink` structures, which in turn point to the actual kernel objects. The interlinking between all the structures is visualized in Figure 2. Kernel objects can be freed even when there are still some weak references. The associated `WeakLinks` are simply invalidated, which makes future upgrades of `WeakRcs` fail. The memory quota spent for kernel objects can be given back to the application immediately when all `StrongRcs` are gone. There is only a small amount of memory still allocated for the `WeakLink`, which we account to the kernel memory of a task object. A task can only hold a limited number of `WeakRcs` and thus can keep only a limited number of `WeakLinks` alive.

Our abstractions must also make sure that a `TempRc` cannot be held when a task is suspended and that a task cannot gain access to a `StrongRc` form a kernel-internal collection. The first invariant is enforced using a global counter that is incremented when constructing a `TempRc` from a `StrongRc` or by upgrading a `WeakRc`. The counter is decremented when destroying the `TempRc` again. When switching tasks, this counter is checked to be zero. For the second invariant, we have modified our kernel-internal collections to only ever give out `TempRcs` although they internally keep `StrongRcs`.

3.2.3 Notification About Object Destruction. The proposed reference counting system can also be used to wake suspended tasks up. The M³ kernel has some system calls that might suspend a task arbitrarily long, for example, decrementing a semaphore or waiting for a child process to exit. These tasks are usually woken up by the expected event, like when the semaphore counter is increased. However, when an associated object is destroyed, the task might never wake up again. For example, destroying the semaphore kernel object would leave the waiting task without any possibility to wake up. Thus, M³ wakes tasks up when associated kernel objects are destroyed. Whenever a kernel task waits for an event, it can also register relevant `WeakRcs`. If one of these `WeakRcs` becomes invalid, the waiting task is woken up. It can now abort the system call as it has become unfulfillable.

4 Evaluation

As our design introduces runtime overhead to enable immediate resource reclamation, our evaluation quantifies this overhead. We start with microbenchmarks that study the impact on system call performance, followed by the overhead for application launches.

4.1 Evaluation Platform

We use the existing FPGA platform of M³, which is implemented on a Xilinx Virtex UltraScale+ FPGA (VCU118 board). We put eight processing tiles with a single RISC-V core each onto this FPGA platform. Additionally, we use two memory tiles to access external DDR4 DRAM. All tiles are connected by a NoC using a 2x2 star-mesh topology. The platform uses Rocket cores and BOOM cores as the RISC-V cores. Rocket is a 64-bit RISC-V in-order core with MMU and 16 kB L1 cache and a 512 kB L2 cache. BOOM is the out-of-order variant of Rocket with the same cache configuration. The clock frequencies of the Rocket and BOOM cores are set to 100 MHz and 80 MHz, respectively, to fully meet timing requirements during FPGA synthesis and place-and-route. The M³ kernel runs on a Rocket core, whereas all benchmarks in this evaluation run on BOOM cores.

4.2 Performance Impact on Syscalls

We start with microbenchmarks to better understand the performance impact of our proposed design. We measure the latency of several system calls supported by M³. In general, system calls on M³ are implemented with message passing. That is, the application marshalls a message, sends it to the kernel, which unmarshalls the message, handles the system call accordingly, and sends a reply to the application. Our benchmark therefore measures the time for this entire procedure for different system calls. To explain the implications of our design, we compare three variants: 1) holding strong references during ASYNC calls, as was done by M³ previously, 2) downgrading to weak references during ASYNC calls, but using the implementation from `std::rc`, and 3) downgrading to weak references using our reference-counting implementation. In contrast to the third variant, the second variant does not allow to free the memory until the last strong *and* weak reference is gone.

We perform each system call 1000 times after 100 warmup calls and show the average latency including standard deviation in Figure 3. As can be seen, most system calls experience

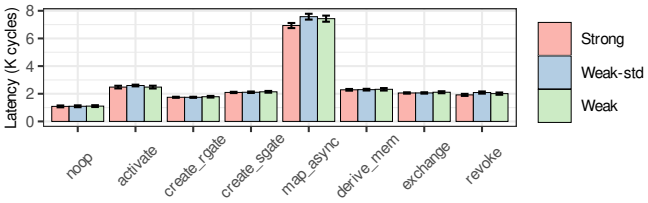

Figure 3. System call latencies with strong references ("Strong"), weak references using `std::rc` ("Weak-std"), and our custom reference-counting types ("Weak").

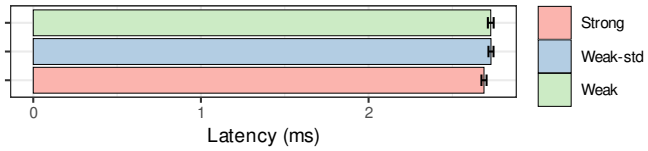

Figure 4. Application start and teardown performance with strong references ("Strong"), standard weak references ("Weak-std"), and custom weak references ("Weak").

a modest slowdown (up to 4.6%). However, the asynchronous `map_async` syscall is 7.2% slower with "Weak" compared to "Strong". The reason is that this syscall involves several `TempRcs` that are downgraded before the ASYNC call to TileMux, which is instructed by the kernel to insert page-table entries. We also observe that the additional allocation that is required for our custom reference counter does not lead to significantly more overhead. In some cases, the "Weak" variant is even faster than "Weak-std", which is due to specific optimizations for the usage within the M³ kernel.

4.3 Application Start and Teardown

After quantifying the overhead for system calls, we asked how much application performance would degrade. We ran a complex leveldb workload – involving pager, filesystem, and network stack – but this lead to almost identical results for all three variants. The reason is the design of M³ where applications access OS services like file systems and network stacks directly via DTUs without involving the kernel. System calls are primarily used during application start and teardown phases to establish the DTU-based communication channels that are used during runtime. For that reason, we decided to measure the impact on application launches instead.

The benchmark creates a new process, establishes memory mappings via the pager for code, data, stack, and heap and starts the application. The application consists of an empty `main` function. The benchmark waits until the application exited and destroys the process afterwards. For that reason, several system calls are required for process creation, communication-channel setup, and their revocation afterwards. We use four runs after two warmup runs and show the results in Figure 4. As can be seen, even for workloads that involve system calls, the performance overhead is relatively low (1.5%).

5 Related Work

To the best of our knowledge, there is no previous work that tries to solve the same problem as this paper. Nevertheless, there is related work that discusses the topic of resource reclamation [4], robust asynchronous code [6], and kernel object lifetimes [3]. Furthermore, there are also well-known related concepts that our approach can be compared to.

Reference Counting and Garbage Collection. Both are memory safe and support weak references. However, they do not enforce immediate resource reclamation and make it easy to keep object references for too long.

Locks. Acquiring a per-object lock when holding a reference is a valid, alternative approach. However, it is not trivial to avoid deadlocks and resource reclamation gets delayed.

RCU. Read-copy-update is a method to avoid locks while threads access data structures in parallel. It is thus orthogonal to our single-thread-focused approach.

No Concurrency. The problems described in this paper do not occur without asynchronous computation and thus without concurrency. For example, *seL4* [5] does not run kernel threads concurrently. Instead, long-running system calls are fully aborted on preemption and restarted afterwards. We do not believe that this programming model can be applied to M³ with its asynchronous communication between the kernel and Tilemux. Dealing with asynchronous calls by suspending tasks seems more ergonomic and easier to comprehend.

6 Conclusion

This work presented an approach for managing object references so that resources can be reclaimed immediately. We enforce that suspended tasks can only hold weak references. The implementation is made robust against programming mistakes using static analysis and runtime checks. An evaluation on the M³ platform showed that runtime performance is only minimally impeded. In future work, we intend to look into how much memory our approach can safe in kernel-resource-contended scenarios. We assume that not only kernels might take advantage of the proposed approach. Whenever services use asynchronous computation and globally-shared objects, robust and immediate resource reclamation seems desirable.

Acknowledgments

This research is funded by the European Union's Horizon Europe research and innovation program under grant agreement No. 101092598 (COREnext).

References

[1] Nils Asmussen. 2024. M3. https://github.com/Barkhausen-Institut/M3

[2] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. 2016. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Atlanta, GA, USA). ACM, 189–203. https://doi.org/10.1145/2872362.2872371

[3] Jongmoo Choi, Seungjae Baek, and Sung Y. Shin. 2006. Design and implementation of a kernel resource protector for robustness of Linux module programming. In *Proceedings of the 2006 ACM symposium on Applied computing*. ACM, Dijon France, 1477–1481. https://doi.org/10.1145/1141277.1141621

[4] Sang-Hoon Kim, Jinkyu Jeong, Jin-Soo Kim, and Seungryoul Maeng. 2016. SmartLMK: A Memory Reclamation Scheme for Improving User-Perceived App Launch Time. *ACM Transactions on Embedded Computing Systems* 15, 3 (July 2016), 1–25. https://doi.org/10.1145/2894755

[5] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP'09)*. ACM, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[6] Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. 2018. Synchronizing the Asynchronous. In *29th International Conference on Concurrency Theory (CONCUR 2018) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 118)*, Sven Schewe and Lijun Zhang (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 21:1–21:17. https://doi.org/10.4230/LIPIcs.CONCUR.2018.21

[7] MDN Contributors. 2024. Introducing asynchronous JavaScript. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing

[8] Samuel Moelius. 2021. Write Rust lints without forking Clippy. https://blog.trailofbits.com/2021/11/09/write-rust-lints-without-forking-clippy/

[9] Taylor Cramer. 2024. async/await. https://rust-lang.github.io/async-book/03_async_await/01_chapter.html

[10] The Deno Authors. 2024. Deno, the next-generation JavaScript runtime. https://deno.com/

[11] The Rust Community. 2024. Borrowing. https://doc.rust-lang.org/stable/rust-by-example/scope/borrow.html

[12] The Rust Project Developers. 2024. Clippy. https://github.com/rust-lang/rust-clippy

[13] The Rust Project Developers. 2024. RefCell in std::cell. https://doc.rust-lang.org/std/cell/struct.RefCell.html

[14] The Rust Project Developers. 2024. Rust Programming Language. https://www.rust-lang.org/

[15] The Rust Project Developers. 2024. std::rc. https://doc.rust-lang.org/std/rc/index.html

[16] The Rust Project Developers. 2024. Weak in std::rc. https://doc.rust-lang.org/std/rc/struct.Weak.html

[17] The Rust Project Developers. 2024. What is rustc? https://doc.rust-lang.org/rustc/index.html

[18] Tokio Contributors. 2024. Tokio - An asynchronous Rust runtime. https://tokio.rs/