Bounded Resource Reclamation

Viktor Reusch
Barkhausen Institut
Dresden, Germany

***Abstract*—Resource allocation is well-studied in operating systems, but resource reclamation remains largely underexplored. This paper investigates the impact of unpredictable resource reclamation latency on system behavior, particularly in resource- and time-constrained environments like Open-RAN and serverless functions. We study scenarios of high reclamation times across various systems. Under adverse conditions, reclaiming resources can delay process termination by multiple seconds on both Linux and the L4Re microkernel. We propose a design for accounting and bounding resource reclamation latency to enable predictable system operation and mitigate potential denial-of-service scenarios. We also advocate for optimizing for the case of bulk reclamation — reducing worst-case reclamation latency by multiple orders of magnitude.**

***Index Terms*—Operating Systems, Real-time systems and embedded systems, Reliability, Allocation/deallocation strategies**

I. Introduction

Resource management is a key concern when seeking *predictable system behavior*. Processes frequently allocate and release resources during their lifetime. Thus, the performance and timeliness of resource allocations through the OS is critical for stable application performance. This includes the consideration of tail latencies and worst-case execution times. Consequently, existing work focuses on improving the performance of allocation operations.

In contrast, the behavior of resource reclamation is understudied. The operating system has to manage a variety of limited resources like user memory (e.g., for heaps), kernel memory (e.g., for page tables), or more specific resources like available TCP port numbers. Because these resources are limited, the OS has to reclaim them to make them available to new allocations. These reclamation operations are especially prevalent during the termination of processes. Reclamations occur in bulk as resources held by the terminated processes are released. This can lead to unpredictable system behavior mainly due to two effects: First, the reclamation operations themselves need to be executed and thus they occupy the CPU. This is often reflected by terminations of processes taking longer than usual. Meanwhile, the operating system collects the released resources. Second, if the whole system is running resource constrained, consecutive allocations (e.g., during startup of a new process) have to wait for resources to be released. Thus, new processes have to wait on the termination of previous processes to fully release resources. In conclusion, the latency of release operations affects the subsequent system behavior causing a threat to predictability.

There are prominent use cases that could benefit from a predictable resource reclamation behavior. Predictability can be ensured by enabling the operating system to proactively enforce a time bound on reclamation operations. We call this concept *bounded resource reclamation* (*BRR*). A first possible use case of BRR is a resource-constrained multi-user system as found in software-defined radio solutions like *Open-RAN*. Such Open-RAN units are numerously deployed in the field with limited system resources due to, e.g., power and cost constraints. Thus, applications of different stakeholders need to share resources. These systems also need to be dynamic, allowing termination and recreation of applications. Resource reclamations of terminating processes could interfere with newly created processes thus delaying application startup. This shows the necessity of BRR in resource-constrained systems. Second, also cloud settings, like function-as-a-service environments, can benefit from bounded resource reclamation. Cloud functions are typically assigned a fixed main memory allocation and a hard time limit. This allows the cloud provider to cost-effectively maximize system utilization. However, this utilization management is in vain if resource reclamation is not considered. As shown in Fig. 1, a rogue function of an untrusted customer could deliberately prolong resource reclamation and thus delay system operation. Again, the concept of BRR is needed to solve this issue.

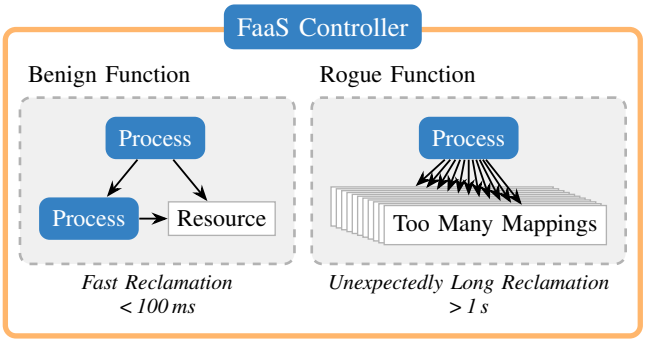

Fig. 1. While most benign functions in a FaaS environment terminate quickly, some other functions might behave unexpectedly and allocate many system resources in rather complicated ways. Reclaiming these resources takes unexpectedly long.

To tackle the issue of unpredictable reclamation behavior, we propose to plan ahead for resource reclamation. We especially focus on the bulk reclamation of resources during application termination. This seems to be the most prevalent scenario, at least in the described use cases. A predictable system should implement accounting of the time needed to reclaim resources and thus allow setting strict time bounds. This enables controller services or orchestrators to guarantee timely reclamation of resources even when untrusted applications exceed their

time budget and have to be forcefully terminated. We also propose to optimize for bulk reclamations by employing arena allocation for management data structures, which drastically reduces the cost of reclaiming whole process trees.

This paper starts with an analysis of the worst-case reclamation time of process resources on various systems (Section II). We present a design of a system that accounts for the latency of resource reclamations (Section III). Additionally, we advertise for grouping resources in prospect of reclamation to accelerate system operation. The implementation of a prototype of this design is then presented in Section IV and evaluated in Section V.

II. Study of Reclamation Latency

To show the necessity for bounded resource reclamation, we examine whether unbounded resource reclamation can lead to unexpected delays in system operation. This section first demonstrates that resource reclamation can actually delay process termination by noticeable amounts. In the examined scenario, a process allocates an unusually large amount of resources in a rather complex way, e.g., by allocating a lot of memory as singular pages. These allocations are correctly accounted towards the CPU time quota of the process. However, when the process gets terminated, e.g., due to exceeding its FaaS time limit, resource reclamation will take unusually long. This reclamation time is not accounted for and might lead to unexpected system behavior by delaying further application startups.

This study of reclamation latency looks at the resource reclamation of kernel memory under three different operating systems: Linux, L4Re [1], and M^3 [2]. On all these platforms, a single process is spawned. It requests the allocation of many page mappings to the same physical page. So no additional user memory is needed to back the page mappings. However, the management structures for these mappings require lots of kernel memory. Later, the process is forcefully terminated. The termination latency is measured to see how resource reclamation prolongs termination latency.

Another, similar benchmark is conducted to assess the potential of the reclamation process to slow down the progress of the whole system. During the termination of the process with the mappings, another workload is started. This workload consists of spawning processes in a tight loop, stressing the kernel subsystems. By measuring how much this workload is slowed down, one can asses how much the operation of a, e.g., FaaS system would be slowed down by long reclamation operations.

Linux is a very common platform for running FaaS workloads, which could experience unbounded resource reclamation. The Linux benchmarks of Fig. 2 are run using kernel version 6.7.4 on an Intel Xeon Platinum 8358 CPU. The governor is set to performance, SMT is disabled, and benchmark programs are pinned to a single CPU core. As the plot on the top left shows, the termination of a *small* Linux process normally takes only around 72.7 µs. However, when the process has one thousand memory mappings, the termination latency already increases to 334 µs. In its extreme, the latency can reach up 13.8 seconds for cleaning up 32 million mappings. Of course, having these many mappings also requires to a lot of kernel slab memory — around 8.9 GiB more. The bottom left plot in Fig. 2 highlights how a workload running during the termination is affected. For this, the plot depicts the runtime of the concurrently-running workload relative to its base runtime in a quiescent system. Thus a relative runtime of one would imply that termination and reclamation have no effect on the workload runtime. When the terminating process only has a single mapping, the concurrent workload is only about 12.2 % slower than its baseline. If the termination latency is high, we see a slow down of up to 115 %. So the workload execution time is roughly doubled. The Linux scheduler seems to equally distribute CPU time between the reclamation operation and the workload. This behavior is, of course, not applicable to all workloads and scheduling strategies but gives one concrete example of termination latency affecting system performance.

The second column of plots in Fig. 2 depicts the results under the L4Re system. The system hardware is the same as previously under Linux. L4Re is a small, microkernel-based operating system with real-time capabilities. Discussing resource reclamation latency on L4Re is not only interesting when discussing real-time but also in the light of recent work that is exploring function-as-a-service workloads on L4Re [3]. In the top L4Re plot, we again see a correlation between mapping count and termination latency. The latency varies from 9.7 ms up to 3.0 s depending on the number of mappings. There is again a big increase in kernel memory consumption of 724 MiB. Concurrent workloads on the L4Re platform are slowed down by the reclamation operations as shown by the lower plot. At 100 000 mappings, the concurrent workload runs 29 % slower. When reclaiming 32 million mappings, the execution time of the workload even becomes 106 times longer. The likely cause of this large slowdown is the *helping* strategy of the L4Re kernel locks. The reclamation code path in the kernel often has to acquire locks. Other kernel threads that try to acquire the already-taken locks will lend the CPU to the lock-holding thread so that it can make progress and release the lock eventually. This leads to the termination operation receiving more CPU time than the concurrent workload.

The third system examined in Fig. 2 is M^3, which is a hardware-software co-design that focuses on security-critical use cases, such as telecommunication infrastructure [4]. M^3 has unique hardware resources, like hardware-based communication channels, that make it interesting for a study on resource reclamation. The results under M^3 are obtained using the gem5 [5] system simulator.[1] Due to the comparatively slow speed of simulation, we limited our M^3 scenario to thousands of kernel structures. The M^3 system not only supports large numbers of memory mappings but also of semaphore kernel objects. Thus, we additionally examined semaphores under M^3. The plots show a similar overall trend for M^3 as already for L4Re. For example, termination latency increases from 313 µs to 16.2 ms when reclaiming 10 000 semaphore objects. Because the count of kernel objects is lower in the M^3

[1] A simulator is necessary because of the custom hardware components in an M^3 system.

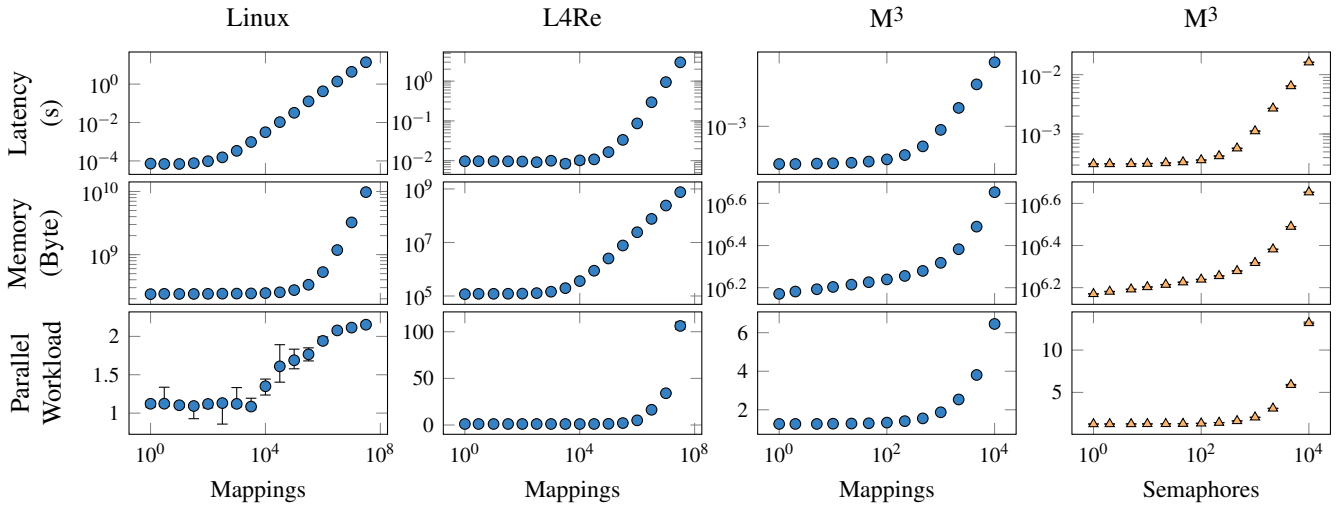

Fig. 2. These plots evaluate our study on reclamation latency. The columns represent different OSes and types of kernel objects. The first row shows the latency of process termination depending on the number of kernel objects allocated. The kernel memory footprint before termination is shown in the second row. The third row shows the relative execution time slowdown of a workload running concurrent with the termination. All depicted values are medians with error bars showing the fifth and 95th percentile. Most axes have a logarithmic scale to better show correlations across orders of magnitude.

benchmarks, the increase in kernel memory consumption is only about 2.88 MiB. Similar to the L4Re scenario, a concurrent workload is heavily slowed down by concurrent reclamations. The likely cause is that the scheduling of system calls in the M^3 kernel does not make fairness considerations.

Overall, this study on termination latency reveals that reclaiming resources — at least in the kernel — can take a considerable amount of time. Of course, allocating high numbers of kernel objects also entails consuming larger amounts of kernel memory. Thus, one could try to limit the latency of memory reclamations by using existing mechanisms for kernel memory accounting, e.g., via Linux cgroups [6]. However, limiting the kernel memory usage of specific processes is only a proxy metric for reclamation time. There is currently no holistic approach for enforcing a time bound on resource reclamation. Thus, the next section will discuss how a system for explicitly accounting reclamation time can be designed. This holistic approach also gives the opportunity to optimize resource management for the case of bulk reclamation, e.g., on process termination. This might be particularly interesting for short-lived function is FaaS settings.

III. Design for Bounded Resource Reclamation

The analysis in the previous section has shown that reclamation latency can be a threat to predictable system behavior. Hence, there arises the need for a design ensuring bounded resource reclamation. As motivated by the use cases, we assume that there is always some controller in the system that creates and terminates (groups of) child processes. This controller is also the entity that should enforce a time bound on the reclamation of resources. For example, a FaaS provider might want to limit the latency of processes termination to at most 100 ms. This allows for predictable high-level scheduling of customer functions. Hence, a controller should be able to set a bound of 100 ms on the reclamation latency of each child

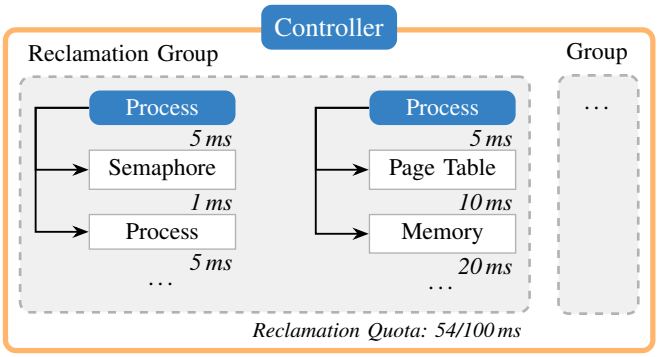

Fig. 3. The abstract design of BRR assigns each reclamation group a reclamation time budget. Whenever a kernel object is allocated, the expected time needed to reclaim this object is subtracted from the group's quota. If the quota reaches zero, no further allocations can be performed.

group of processes. Reclamation latency thus has to become a virtual resource on its own with budgets and accounting.

For the simplest form of BRR, the controller would assign every (group of) processes with a specific reclamation time budget. Then, whenever a resource is allocated the OS checks if this resource will be reclaimed when the group is terminated. For such allocations, the OS subtracts the anticipated reclamation time from the group's quota. The combined reclamation latency of all allocations made by the group must thus not exceed its budget. We call a group of processes and resources sharing the same reclamation quota a *reclamation group*. An example of this approach is shown in Fig. 3. In summary, the OS accounts for the reclamation time of processes in advance.

Of course, there are some complications in real-world operating systems. Most prominently, processes can nest by forking of child processes. These should also be subject to the same budget constraints. Thus, reclamation budgets have to be inherited.

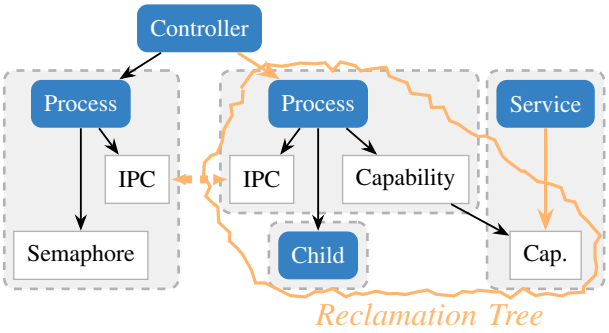

Fig. 4. Whenever a process group or a process hierarchy is terminated, many kernel resources get freed. This includes child processes and derived capabilities. All these released kernel objects form the reclamation tree. There are also connections from the outside to objects inside the reclamation tree, e.g., IPC channels to system services.

Furthermore, the reclamation of a process might trigger the deletion of other OS objects like network sockets or capabilities. Capabilities on microkernels, like L4Re and M^3, are especially interesting as they can be delegated and derived. This essentially creates inheritance trees inside the kernel that also need to be honored for BRR. Overall, these interdependencies in the different OS objects mean that upon termination of a process, a whole subtree of objects will be reclaimed. We call this subtree the *reclamation tree* starting from some root process. An example is given in Fig. 4. The reclamation latency of all the objects in the reclamation tree of a reclamation group has to be accounted in the associated reclamation quota for accurate accounting.

The observation of the reclamation tree leads to a possible optimization. For the use cases of FaaS and Open-RAN, one can assume that there are few interconnections between the reclamation tree of a customer process and the rest of the system. These interconnections are likely limited to user memory and networking. Presumably, the bulk of reclamation operations for these customer processes are internal to the reclamation tree. Examples for such reclamation operations are deleting user memory mappings, terminating inter-process-communication channels, or deallocating process objects in the kernel. These cleaned-up objects are only linked internally with respect to the reclamation tree. So all objects in the reclamation tree will be reclaimed during a bulk reclamation. Henceforth, there is actually no need to unlink individual objects. Both sides of these object-to-object links will be freed and reclaimed. Thus, only interconnections to the rest of the system that link outside of the reclamation tree need to be unlinked.

Going a step further, one can optimize the deallocation of all these objects inside the reclamation tree by using some form of arena allocator for all objects inside the reclamation tree. The reclamation group would get a single, large chunk of memory (the arena) in advance. Afterwards, all objects in the reclamation tree of the group are allocated in memory inside of the arena. This grouping of objects allows the reclamation of a whole tree by simply reclaiming one large chunk of memory and unlinking a couple of outside interconnections — opposed to deallocating each object individually. This optimization should greatly reduce reclamation time and thus help to keep bounds on reclamation time low.

IV. Prototypical Implementation

To test the effectiveness of our approach, we have implemented a prototype of bounded resource reclamation. We chose the M^3 operating system as our implementation platform. M^3's microkernel design fits well to the idea of a reclamation tree. Kernel objects and capability inheritance already span a graph inside kernel memory. Furthermore, M^3 is a hardware/operating-system co-design with interesting hardware resources that need to be considered during reclamation. The fundamental design idea of M^3 is to isolate individual CPU cores using custom hardware-based isolation units. This tiled architecture already sketches boundaries for process groups that can be leveraged for BRR. In this first prototype, we limit our implementation to the grouping mechanism in the kernel itself including arena allocation. The actual accounting of the reclamation time, especially for connections leading outside of the reclamation tree, will be added in future work.

First, one has to consider how a reclamation tree looks like in the M^3 kernel. The M^3 microkernel is capability-based and thus the kernel manages a list of capabilities for each process. This means that processes can only access the kernel functionalities/abstractions they have a capability to. In the kernel, these capabilities point to kernel objects like semaphores, process structures, or page mapping entries. Processes on M^3 can collaborate by exchanging capabilities pointing to these objects. Thus, an inheritance graph of capabilities and kernel objects is created. When a process is terminated and the process structure is reclaimed in the kernel, all capabilities held by the process will be freed as well. Consequently, all inherited capabilities (even when exchanged with other processes) are reclaimed. Kernel objects that are no longer referenced by any capability are reclaimed. Because kernel objects can be process structures themselves, reclamation can recurse into child processes. This can lead to the cleanup of whole process trees in a single system call under M^3.[2] All this recursion has to be reflected in the reclamation tree of processes and thus has to be respected by our implementation of BRR.

In this implementation of bounded resource reclamation, the kernel enables controllers to assign processes to reclamation groups. Whenever then such a process creates or inherits capabilities/kernel objects, the group affiliation is inherited too. This ensures that all objects in the reclamation tree are assigned to the same reclamation group and can thus be accounted for. Additionally, the kernel allocates each object belonging to a group in a group-local memory arena as shown in Fig. 5.

Some objects in the reclamation tree might also point outside of the reclamation group. These could be, e.g., semaphore objects that are shared with an outside, system-wide network service. The BRR implementation addresses these outside connections by referencing the offending objects in a group-local *scrub list*. When reclaiming a group, the objects in the

[2]This is similar to what can be achieved with PID namespaces under Linux.

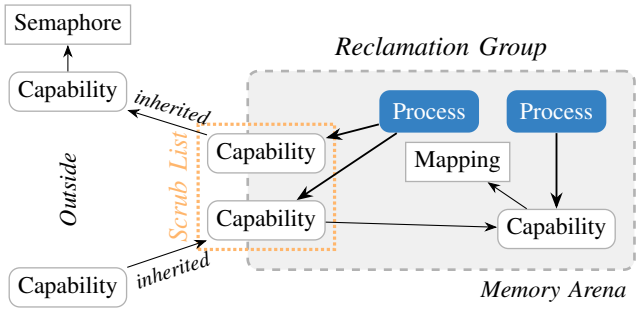

Fig. 5. Kernel objects of the same reclamation group are allocated together inside the same memory arena to optimize for the case of bulk reclamation. Connections leading out of the reclamation group need to be handled individually and are thus noted in a scrub list.

scrub list are unlinked first before the whole arena allocation of the group is deallocated at once. The individual objects inside of the group do not need to be individually unlinked or deallocated, which greatly reduces reclamation latency.

In the future, the arena allocation and each element in the scrub list need to be accounted for in a reclamation time quota. Furthermore, the implementation is currently only fully realized for semaphore kernel objects — just to serve as a proof-of-concept. Nevertheless, we think this implementation can serve for an initial evaluation of the feasibility and effectiveness of the approach as shown in the next section.

V. Evaluation

For the evaluation, we perform measurements on M³ using the gem5 simulator for RISC-V. We compare three different system configurations. First, we measure on a baseline M³ system that does not contain any custom modifications for BRR. Second, we use a modified kernel that implements the changes outlined in the previous section. However, the processes under test are not added to any reclamation group. The results of this configuration show the overhead of the performed code changes to the overall system. For the third configuration, the created processes are actually put inside of a reclamation group to take advantage of the reclamation optimizations.

Figure 6 shows the influence of the modifications on the latency of application startup. The median startup latency increased by only 4.69 µs or 0.81 % from a baseline of 581 µs due to the modifications to the M³ kernel. The increase in latency is higher when starting up the process inside of a group. It increases by 15.3 µs or 2.6 %. This shows that the overhead for the additional bookkeeping of kernel objects inside of reclamation groups and additional branch instructions in system call handlers is small but noticeable.

The overhead of the M³ modifications are also noticeable when creating new kernel objects via system calls. Figure 7 shows that the overhead of grouping is 377 ns or 3.8 % from a baseline of 9.9 µs for creating a single memory mapping. The overhead is more pronounced when creating a semaphore object in the kernel. With the baseline being 3.10 µs, the increase in latency is 1.20 µs or 39 %. This shows that the

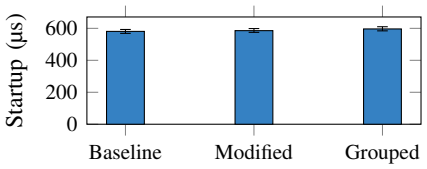

Fig. 6. This plots shows the latency of application startup under an unmodified version of M³, the modified one, and the modified one when starting the process inside of a reclamation group.

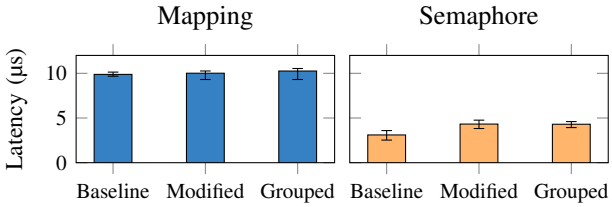

Fig. 7. This shows the latency of allocating a single kernel object under the three tested system settings.

modification to the kernel, which are needed for bounded resource reclamation, can have an effect on short, simple system calls. However, this effect disappears for more involved operations like application startup in Fig. 6.

The preliminary implementation of BRR is already able to drastically speed up termination as shown in Fig. 8. The left most data points show the termination latency of a process with only a single semaphore object allocated inside the kernel. The measured latency is very similar between not using reclamation groups (398 µs) and using the feature (370 µs). Without the group feature and with 10 000 allocated semaphores, the reclamation latency reaches 17 ms. However, with the group feature, the latency increases by only 12 % to 414 µs. This is achieved by allocating kernel objects of a single reclamation group inside a memory arena and freeing them in bulk. The semaphore objects allocated for Fig. 8 do not have any outside connections and can therefore be freed without unlinking individual outside connections. In contrast, the figure clearly shows that without using groups, the reclamation latency increases as all semaphore objects need to be cleaned up individually. Overall, our preliminary implementation already highlights the potential time savings that are possible when optimizing for the case of bulk resource reclamation as often seen in time-constrained FaaS settings.

VI. Discussion

The proposed grouping technique — as a sideeffect — makes memory accounting in the kernel easier and more precise. Normally, kernel memory budgeting can only be an estimate because of fragmentation between different-sized slab allocators and general kernel allocators. Because of this fragmentation, a kernel, like the L4Re microkernel, could run out of memory even though no processes exceeds its kernel memory budget. This imprecise accounting could be avoided by using the proposed grouping approach using an

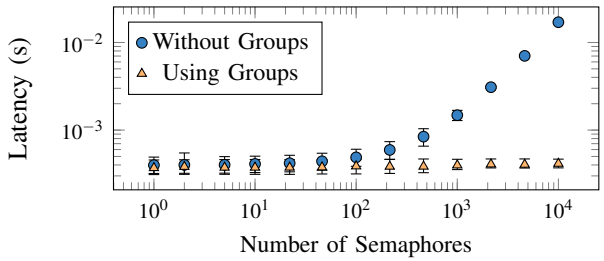

Fig. 8. This plots depicts the time needed to terminate a process depending on the number of semaphore objects it allocated via the kernel. The usage of reclamation groups keeps the termination latency low even for high numbers of objects.

arena allocator. All memory is accounted for in advance and fragmentation issues are contained to the individual groups. Although this comes at the cost of flexibly and dynamically using kernel memory, one could argue that this flexibility is not needed in settings like FaaS. Resources are typically assigned in advance to the various customer's processes to allow partitioning the multi-tenant cloud machines.

Though, there are challenges when elevating BRR to the whole operating system. System services also need to reclaim resources once client processes terminate. This includes network services closing sockets and memory services freeing user memory. One possible solution could include some interplay between OS services and the kernel to manage reclamation time quotas. Alternatively, the controller could allocate separate reclamation budgets at every service. To keep the first prototype simple, this paper focuses on the kernel-internal reclamations of kernel memory.

VII. Related Work

To the best of our knowledge, there is no previous work about a holistic approach to account for reclamation time and optimize for it. Nevertheless, there are related works that partially overlap with the goals of bounded resource reclamation.

Blackham et al. [7] analyze the worst-case execution time of operations in the seL4 kernel. They make sure that non-interruptible sections in the kernel are bounded — thus making it feasible to swiftly react to interrupts. A similar timing analysis would also aid BRR as it is important to estimate the latency of individual reclamation operations. The analysis of Blackham et al. does not put a bound on the latency of reclamations as a whole, only on the individual non-interruptible sub-operations of which there can be unboundedly many. This gap can be filled with by the described design of bounded resource reclamation.

There are quite a few works that are concerned with real-time memory management using dynamic garbage collection [8], [9], [10]. This also entails reclaiming memory with predictable latency to make it available for future allocations. For example, Baker [8] designs a real-time garbage collector that reclaims memory during subsequent allocations. Thus, system progress is not stalled when releasing memory or during periodic scans. Every operation is rather bounded by constant time. However, it is unclear how this concept could be practically transferred to operating systems. First, resources need to be handled by a global garbage collection algorithm. This does not fit the common design of manual resource management typically found in OS kernels. Second, application performance would depend on the amount of resource reclamations that have to happen during allocation operations. Thus, the performance of, e.g., FaaS functions would still depend on the behavior of the previous tenant of the system — just with better predictable time bounds. In contrast, bounded resource reclamation eliminates this correlation by reclaiming all resources directly on termination in bounded time.

The *Thundering Herd* attack by Mergendahl et al. [11] is an example of why predictable system behavior is important. Mergendahl et al. are concerned with attacks that use the kernel in unusual ways to introduce unexpected timing behavior in victim threads. The described attacks make use of the inner workings of the seL4 scheduler implementation to break temporal isolation. This allows many low-priority threads to arbitrarily delay scheduling of a high-priority thead.

Furthermore, there are existing OS mechanisms to enforce restrictions on resource consumptions. Both Linux' *cgroups* and L4Re's *factories* allow processes to restrict (kernel) memory consumption of children. However, simply restricting the maximum allocation of memory does not serve as an adequate proxy for limiting reclamation latency on termination. For example, the cgroup owning shared memory is in-deterministic when multiple cgroups are involved [6]. The factory abstraction in the L4Re microkernel is only concerned with accounting kernel memory. Thus, memory consumption in services on behalf of applications needs to be handled separately. In general, using existing memory accounting to enforce a bound on reclamation latency is imprecise as the latency to reclaim different objects varies. For example, reclaiming a single, large allocation for a task stack might be a lot faster than reclaiming individual, small semaphore objects. BRR offers a holistic solution by accounting for reclamation latency in advance.

VIII. Conclusion and Future Work

This paper has demonstrated the critical impact of resource reclamation latency on system predictability. Our study showed that uncontrolled reclamation can significantly prolong process termination. The observed delays under Linux, L4Re, and M^3 underscore the need for proactive management of reclamation behavior. Our proposed design, focused on accounting and bounding reclamation latency, allows for more reliable systems. Additionally, the possibility of optimizing for bulk reclamations emerges with the potential for orders-of-magnitude latency reductions.

The current implementation is only in a prototypical state with lots of improvements for future work. We would like our implementation to support the actual, precise accounting of reclamation time and to work with more kinds of kernel objects (especially memory mappings). In the end, BRR needs to be expanded to the whole operating system, including services, to fully bring predictable reclamation behavior to real-world scenarios.

REFERENCES

[1] A. Lackorzynski and A. Warg, "Taming subsystems: Capabilities as universal resource access control in L4," in *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems, IIES '09, Nuremburg, Germany, March 31, 2009*, M. Engel and J. Nolte, Eds., ACM, 2009, pp. 25–30. DOI: 10.1145/1519130.1519135. [Online]. Available: https://doi.org/10.1145/1519130.1519135.

[2] N. Asmussen, M. Völp, B. Nöthen, H. Härtig, and G. P. Fettweis, "M3: A hardware/operating-system co-design to tame heterogeneous manycores," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016*, T. Conte and Y. Zhou, Eds., ACM, 2016, pp. 189–203. DOI: 10.1145/2872362.2872371. [Online]. Available: https://doi.org/10.1145/2872362.2872371.

[3] T. Miemietz et al., "A perfect fit? - towards containers on microkernels," in *Proceedings of the 10th International Workshop on Container Technologies and Container Clouds, WOC 2024, Hong Kong, Hong Kong, December 2-6, 2024*, ACM, 2024, pp. 1–6. DOI: 10.1145/3702637.3702957. [Online]. Available: https://doi.org/10.1145/3702637.3702957.

[4] S. Haas et al., "Trustworthy computing for O-RAN: security in a latency-sensitive environment," in *IEEE Globecom 2022 Workshops, Rio de Janeiro, Brazil, December 4-8, 2022*, IEEE, 2022, pp. 826–831. DOI: 10.1109/GCWKSHPS56602.2022.10008543. [Online]. Available: https://doi.org/10.1109/GCWkshps56602.2022.10008543.

[5] N. L. Binkert et al., "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011. DOI: 10.1145/2024716.2024718. [Online]. Available: https://doi.org/10.1145/2024716.2024718.

[6] T. Heo. "Control group v2," The Linux Kernel documentation, Accessed: Apr. 16, 2025. [Online]. Available: https://docs.kernel.org/admin-guide/cgroup-v2.html.

[7] B. Blackham, Y. Shi, S. Chattopadhyay, A. Roychoudhury, and G. Heiser, "Timing analysis of a protected operating system kernel," in *Proceedings of the 32nd IEEE Real-Time Systems Symposium, RTSS 2011, Vienna, Austria, November 29 - December 2, 2011*, IEEE Computer Society, 2011, pp. 339–348. DOI: 10.1109/RTSS.2011.38. [Online]. Available: https://doi.org/10.1109/RTSS.2011.38.

[8] H. G. Baker Jr., "List processing in real time on a serial computer," *Commun. ACM*, vol. 21, no. 4, pp. 280–294, 1978. DOI: 10.1145/359460.359470. [Online]. Available: https://doi.org/10.1145/359460.359470.

[9] H. Lieberman and C. Hewitt, "A real-time garbage collector based on the lifetimes of objects," *Commun. ACM*, vol. 26, no. 6, pp. 419–429, 1983. DOI: 10.1145/358141.358147. [Online]. Available: https://doi.org/10.1145/358141.358147.

[10] P. Cheng and G. E. Blelloch, "A parallel, real-time garbage collector," in *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, M. Burke and M. L. Soffa, Eds., ACM, 2001, pp. 125–136. DOI: 10.1145/378795.378823. [Online]. Available: https://doi.org/10.1145/378795.378823.

[11] S. Mergendahl, S. Jero, B. C. Ward, J. Furgala, G. Parmer, and R. Skowyra, "The thundering herd: Amplifying kernel interference to attack response times," in *28th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2022, Milano, Italy, May 4-6, 2022*, IEEE, 2022, pp. 95–107. DOI: 10.1109/RTAS54340.2022.00016. [Online]. Available: https://doi.org/10.1109/RTAS54340.2022.00016.