Group-based Memory Management in Fyr

Torben Weis, Peter Zdankin, Oskar Carl, Marian Waltereit
Distributed Systems
University of Duisburg-Essen
{firstname}.{lastname}@uni-due.de

***Abstract*—Albeit being introduced decades ago, C and C++ are still the most commonly used programming languages for operating systems. These languages have no reliable mechanisms to deal with memory safety issues, such as use-after-free or data race conditions, that are a leading cause for security bugs in operating systems and other critical software. Tools such as Valgrind have been developed to identify errors, but the errors must occur during the analysis, as they are not found otherwise. Several modern programming languages such as Rust, Go and Swift have emerged aiming to solve some of the issues by providing memory safety guarantees at compile or run time. However, these languages introduce new limitations, especially concerning software development for performance-critical or resource-constrained systems. In this paper, we introduce a new approach to automatic memory management that manages the lifetime of object groups instead of individual objects. We show that group-based memory management can remove some of the restrictions of modern programming languages while satisfying important memory safety constraints. Furthermore, we show how group-based memory management is implemented in our new systems programming language Fyr.**

I. Introduction

System software and software for embedded systems is still dominantly developed in C/C++ and a large fraction of programming errors in these languages can be attributed to incorrect memory management. This is expressed in findings like the fact that memory safety issues contribute to 70% of all security bugs in Google Chrome [3]. A reason for this is the flexibility of pointers in C/C++, which can refer to arbitrary memory locations, without any built-in mechanisms enforcing proper allocation and initialization of these memory regions. The most prominent sources of errors are *use after free*, *memory leaks* and *race-conditions*. These errors can either cause crashes, nondeterministic behaviour or drain the available memory due to a failure to free allocated memory.

Tools like Valgrind can be used to detect such errors. However, these tools augment the software and are thus difficult to use for system software or embedded systems, which are very resource-constrained. Furthermore, tools like Valgrind can only detect errors that have happened, such as in test cases. If the test coverage is low, tools like these are of little help.

The other approach is the use of a memory-safe language. Here the safety is derived from compile time checking instead of test time checking. Several modern languages such as Rust, Swift and Go have been positioned as replacements for C. However, except for Rust, these languages are not commonly used for (productive) embedded or operating system development. Rust is currently considered to be used inside the Linux Kernel, is one of the languages that are approved throughout the Fuchsia OS Platform Source Tree, and is supporting many embedded targets through the LLVM backend [15, 13, 16]. In a simple benchmark suite comparing different languages across multiple algorithms, we can observe that Swift achieves execution times comparable to C in some cases, but has usually much higher memory consumption [18].

In this paper, we discuss the shortcomings of these languages when applied to (embedded) system software development, and propose a new memory management scheme to overcome some of these weaknesses. Our approach can be classified as an extension of the principle memory management concept used in Rust.

A major inconvenience in Rust is the strict management of lifetimes for individual, possibly connected, objects. Objects in Rust are deallocated at the earliest possible point in time. If two objects with different lifetimes are added into one data structure, the lifetime of the entire data structure is bound by the shortest lifetime of all objects involved.

We propose to manage the lifetime of groups of objects instead of tracking the lifetime of each individual object. This approach is less restrictive than the borrow checker used by Rust and it can potentially reduce the overhead of memory management. If we have a data structure of objects with different lifetimes that usually would be deallocated at different stages, these objects are now in a group as of our proposal and will be deallocated when the complete group can be freed.

This highlights another optimization goal besides safety and efficiency: expressiveness. A language can be memory-safe and at the same time not allow for data structures such as graphs. For example, the borrow checker used in Rust is essentially only able to check tree-like data structures. It is possible to create graph-like data structures, but the nature of these structures requires more complex solutions that rely on run time reference counting and explicitly unsafe code. A memory-safe replacement for C/C++ should thus cause as little overhead as possible and at the same time give the developer the freedom to express his algorithms and data structures with a minimum of language-imposed constraints and without sacrificing the safety guarantees.

In this paper, we show how group-based memory management can provide advances in this direction. Also, we show how the idea has been implemented in a new systems programming language called Fyr. A new language is neces-

sary because group-based memory management is not a drop-in replacement for other memory management techniques. Instead, it requires special language constructs.

II. Memory Management

In this section, we show how important it is to study how comprehensive the language-based safety guarantees are.

Memory management of programming languages can generally be divided into *manual* and *automatic* memory management. However, the lines between both are blurry and there are various approaches to automatic memory management.

Most system programming languages offer a mix of manual and automatic memory management. For example, C++ offers manual memory management, but the use of *smart pointers* can give developers the feeling that automatic memory management is in place because smart pointers control the lifetime of the objects they point to. However, the compiler does not enforce that smart pointers are used everywhere, which means that there are no safety *guarantees* provided by the compiler. Furthermore, smart pointers add a small performance penalty due to reference counting or the passing of ownership. Additionally, it is always possible to cast smart pointers to raw pointers and thus invalidate all safeguards provided by these constructs.

Languages with automatic memory management by default usually feature a mode to interoperate with C code or to bypass language restrictions for the sake of maximum performance. To do this, languages such as Go, Swift, and Rust introduce the notion of manual memory management or an *unsafe* mode.

Consequently, it is important to evaluate which parts of system software can actually benefit from automatic memory management and which cannot.

A. C Modifications

Solutions that modify C/C++ to achieve memory safety have exhaustively been researched. Several projects extended C with a combination of compile-time and run time checks such as Cyclone [11, 10], CCured [14] or Microsofts CheckedC [19]. Although this paper does not focus on thread safety, it is worth noting that CheckedC, CCured and Cyclone do not offer solutions for thread safety. In contrast, Fyr is thread-safe, because it avoids that one group of objects is concurrently modified by two threads.

SAFE-C [20] is a C dialect that uses run time checks to gain memory safety, but this incurs a performance penalty.

Furthermore, several linters have been developed such as LCLint [8], Metal [7, 6], SLAM [1], PREfix [2], and CQual [17]. Linters are useful helpers to detect common pitfalls, but they cannot prove the absence of a certain error class.

Microsoft VCC [4] (a verifier for concurrent C) is essentially annotated C as well. The annotation used by VCC can come in the form of pre/postconditions and invariants. The correctness is then determined using an automated SMT solver. While VCC is very powerful and was used to verify Hyper-V, its use requires expertise in constraint languages, and sometimes the verifier will diverge, which requires a rethinking of the pre/postconditions. Hence, it is only viable in very critical pieces of software that are already written in C.

B. Automatic Memory Management

Modern systems programming languages dominantly employ automatic memory management. The most prominent mechanism for automatic memory management is *garbage collection*, but it is problematic for time-critical system software and embedded systems. The performance of garbage collectors (GC) is constantly improving, as can be seen by the progress made on the Go GC [12]. However, the timing of the garbage collection runs is non-deterministic. This may not be a problem for typical server software but can have noticeable impacts on interactive applications and high-performance networking software [5]. The second problem is that garbage collection requires more RAM as it slows down under memory pressure due to frequent runs of the garbage collector. On embedded systems with constrained resources, RAM is often limited, which means that GC is no suitable approach.

Another mechanism for automatic memory management is *automatic reference counting* (ARC) as used by Swift. With ARC individual objects are subject to reference counting. The compiler automatically generates code that performs the required reference counting. While the effect is similar to the use of smart pointers in C++, ARC *guarantees* that reference counting is performed correctly. However, reference counting can lead to circular references, i.e. objects pointing to each other in a circle. In such a case *use after free* errors are avoided, but memory leaks are still possible. This means that Swift had to introduce notions of strong and weak references, to circumvent this problem. Furthermore, multi-threading requires the use of *atomic* reference counting, which can have detrimental effects on performance when multiple cores perform atomic reference counting on the same cache line concurrently. Thus, reference counting comes with some run time overhead.

The memory management used by Rust promises *zero overheads* and focuses on the concept of *ownership*. In essence, the compiler ensures that an object can only be *owned* by one pointer, which determines the lifetime of the object. As soon as the variable goes out of scope, the value that is owned by it is dropped. But ownership alone is too restrictive, as objects would have to be moved back and forth whenever they are passed as arguments to a function. Therefore, Rust introduced the concept of *borrowing* an object, which is comparable to pointers in C and references to an object in C++. Borrows are bound by very specific rules: there can only ever be one mutable reference *or* any number of read-only references at the same time. This way the compiler can track when an object has been borrowed and when control of the object returns to its owner. Another rule dictates that a borrow shall never outlive the owner, thus it can be statically proven that references always point to live data and that no data races can possibly exist.

The appealing idea of this approach is that it does not inflict any run time overhead, because borrowing and passing of ownership is only analyzed at compile time and no additional run time actions are required. Hence, the performance of Rust should potentially reach that of C.

Since Rust's memory management is built on ownership and temporary borrowing, only tree-shaped data structures are allowed. This restriction is required, because borrowing is bound to program flow, e.g. a scope or a function call. Borrowed pointers cannot be stored on the heap and outlive the control flow that caused the borrowing. Even a dynamic doubly-linked list cannot be easily represented in Rust for this reason.

One way to bypass this restriction is the use of C++-like smart pointers, which use unsafe Rust internally. Building a doubly linked list this way is possible, but in this case Rust uses reference counting similar to Swift and ensures at run time that *use after free* is avoided.

If data structures in Rust contain references to other objects, they must have a lifetime associated to it, to prohibit invalid access to references. Graph-like data structures can contain references to values on the stack or the heap and thus the complete graph is bound to the shortest lifetime. If we instead elevate these short-lived objects to match the lifetime of the rest of the group, we can have memory safety and expressiveness.

C. Group-based Memory Management

The automatic memory management mechanisms discussed previously try to accomplish the same task as manual memory management, e.g. to free individual objects that are no longer required.

This does not take into consideration that many data structures are built incrementally, but are later freed together. For example, consider an object that represents an HTTP request on the server. During processing the request object grows continuously by attaching more objects for URI, user credentials, tokens, header fields, and response body. When the request finishes, all the aforementioned objects become useless and will be freed.

When the previously discussed mechanisms are in place, the programmer must explicitly tell the compiler that the pointers connecting these objects are alive, i.e. that they do not point to memory that has since been freed. This may even require useless reference counting to ensure the lifetime of said objects is long enough.

The idea of group-based memory management is to exploit the fact that groups of objects grow and are released together. In this case, the compiler (and therefore the programmer) does not have to worry about pointers between these objects since they belong to the same group and therefore the objects are released all at once instead of individually.

To the best of our knowledge, our publication of Fyr was the first to introduce the notion of group-based memory management in programming languages [21]. Later, Microsoft released a new programming language called *Verona*[1]. Verona uses the term *region* instead of *group* but the fundamental idea is similar.

One fundamental difference between Verona and Fyr is the embedding in the programming language. Verona requires the programmer to inform the compiler about groups while Fyr mostly automates the process.

The Verona documentation shows how to allocate two objects in the same region (or group):

```
var x = new Node;
var y = new Node in x;
```

This shows how the Verona programmer indicates explicitly that y should be allocated in the same group as x.

Fyr uses a more convenient approach to determine which objects belong to the same group. The compiler infers groupings by analyzing program flow.

Listing 1: Infer groupings through program flow

```
1 var x = new Node
2 var y = new Node
3 x.next = y
4 y.prev = x
5 return x
```

Using the small section of Fyr code in listing 1 as an example, the process is as follows: In lines 1 and 2 the compiler believes that x and y are two pointers to individual Node objects on the heap, thus each of these belongs to its own group. In line 3 the compiler sees that the object pointed to by x now includes a pointer to the object pointed to by y. This implies that both must belong to the same group, even though the programmer did not declare this manually.

This analysis happens at compile time. Consequently, the compiler generates code for lines 1 and 2 with the knowledge that both objects belong to the same group and thus have the same lifetime. This allows saving a call to `malloc()`, because the compiler can allocate the memory for both objects in a single call to `malloc()`. When pointers x and y go out of scope, the Fyr compiler sees that no pointers to this group remain on the stack frame and frees the memory. This illustrates how group-based memory management can lead to more efficient code by allocating and freeing objects in a group simultaneously.

In line 4 of the example above, the y-object is set to point to the x-object, creating circular references. This is not be allowed in Rust, as ownership cannot be determined since the objects point towards each other. In contrast to this, Fyr understands that the x- and y-objects are already in the same group and they can have arbitrary pointers to each other. This illustrates that group-based memory management can offer more freedom to the programmer to shape data structures as he wishes.

However, not everything can be determined automatically. Each function in Fyr must declare how its parameters are grouped. For example in

[1]https://github.com/microsoft/verona/blob/master/docs/explore.md

```
func f(a *Node, b *Node) { }
```

the function f assures that a and b are pointers and the objects they point to may belong to different groups. Therefore, the function must not force them into one group by connecting the objects in any way. The following lines do therefore cause a compiler error in line 2:

```
func f(a *Node, b *Node) {
    a.next = b
}
```

Expanding the function signature fixes this issue:

```
func f(a 'g *Node, b 'g *Node) {
    a.next = b
}
```

The compiler is instructed that arguments to `f` must be of the same group by annotating both parameters with the same group-specifier `'g`. This is the only case in Fyr where the programmer has to explicitly declare groupings. Everything else is automatically inferred by the compiler.

When calling this function `f`, the compiler uses the function signature to reason about the grouping. In the following example, the compiler infers that `x` and `y` point to objects in the same group, because the signature of `f` demands it.

```
var x = new Node
var y = new Node
f(x, y)
```

If more complex control flow is used, the group analysis becomes significantly more challenging. The code in listing 2 shows a case where the control flow cannot be determined statically.

Listing 2: Static grouping is not always possible

```
1 var x = new Node
2 var y = new Node
3 if luck() {
4     f(x, y)
5 }
6 return x
```

In this case, the two objects belong to the same group if and only if the if-clause is executed. However, this is only known at run time. Due to this, the compiler allocates the x-object and the y-object individually on the heap and assumes that they form two independent groups. If the if-clause is executed, it merges both groups together during run time. Since releasing of memory is done per group, after line 6, the y-object will be freed if and only if the if-clause did not run. While merging groups at run time, the memory allocation system maintains a single linear list per group. This incurs a small run time overhead but simplifies the process of releasing memory.

While a full explanation of Fyr is beyond this paper, we also want to mention that Fyr supports the concept of *foreign group pointers*. These pointers point to objects in other groups and allow the creation of dynamic data structures that can grow and shrink dynamically. Foreign group pointers denote ownership of the other group. In this case, Fyr manages groups with ownership and borrowing somewhat similar to how Rust manages individual objects, while still keeping its flexibility for data structures inside one group.

The language implementation is open-source and still under development. It can be retrieved from the Fyr Git repository [9].

III. Fyr Compiler Internals

The process of detecting these groupings consists of two distinct steps and is the most complex part of the compiler: *Group analysis* is the phase of determining which objects belong to the same group and *group checking* ensures that the program does not try to merge groups that must not be merged.

Analyzing grouping based on the *abstract syntax tree* (AST) of the source code directly is far too complex. Instead, we first compile the Fyr code into *single-static assignment* form (SSA), where each variable is assigned only once. This simplifies the task of tracking groupings significantly and is a common step in modern compilers.

In the code shown in listing 2, SSA turns `y` into a *phi-variable*. A phi-variable is assigned only once and its value depends on the control flow. In Fyr, `y` is associated with a *phi-grouping*. That means the compiler generates a pointer that tracks at run time to which group `y` does belong, which allows for modification of the group at run time. This is required because — without any analysis of the `f` function — the actual group can only be known at run time. This way the compiler provides an automated framework allowing for safe memory usage in scenarios where the exact compositions of objects cannot be determined at compile time.

While this does generally introduce execution overhead at run time, there is huge potential to eliminate run time group merges using more extensive compile time analysis. In Listing 3 no run time costs are inflicted, because the compiler can determine at compile time that `y` is not used outside the if-clause.

Listing 3: No need to merge

```
1 var x = new Node
2 if luck() {
3   var y = new Node
4   x.next = y
5 }
6 return
```

If `y` is ever assigned an object, it belongs to the same group as `x`. This also means that the compiler does not need to generate any code to free the y-object explicitly because it is freed together with the x-object in line 6 when `x` goes out of scope. Since `y` has already left its scope and `x` is the only active object left in the group, the compiler can automatically free all elements in the group at once.

As a compiler backend, Fyr currently produces C99 code that is then compiled with gcc or clang. The advantage of this approach is that we can use the normal tooling to generate code for embedded use cases. At the time of writing, Fyr can produce code for Intel x86-64 and Atmega328 (i.e. Arduino boards). Other build targets should be easy to add

since existing C-compilers can be used and the configuration backend to integrate them is flexible.

IV. Conclusions and Outlook

We have shown that group-based memory management can improve on current mechanisms for automatic memory management by checking the lifetime of groups of objects instead of checking the lifetime of each object individually. This approach can support a broader range of data structures without incurring significant run time overhead.

We presented the embedding of our memory management scheme in the Fyr programming language. As the implementation of Fyr matures, usage of the language will provide feedback on the suitability of group-based memory management in real-world system software. Furthermore, we will try to leverage more of the performance optimizations that become possible with group-based memory management.

With respect to backends, an LLVM backend is a desirable next step to improve the speed of compilation. Work on an experimental Vulkan backend has already started.

Group-based memory management could be utilized to address further systems programming challenges. For example, our approach could be used to target non-homogeneous memory architectures by attaching groups to different memory types such as RAM, Flash, or shared memory region.

References

[1] Thomas Ball et al. "SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft". In: *Integrated Formal Methods*. Ed. by Eerke A. Boiten, John Derrick, and Graeme Smith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1–20. ISBN: 978-3-540-24756-2.

[2] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. "A Static Analyzer for Finding Dynamic Programming Errors". In: *Softw. Pract. Exper.* 30.7 (June 2000), pp. 775–802. ISSN: 0038-0644. DOI: 10.1002/(SICI)1097-024X(200006)30:7⟨775::AID-SPE309⟩3.0.CO;2-H. URL: http://dx.doi.org/10.1002/(SICI)1097-024X(200006)30:7%3C775::AID-SPE309%3E3.0.CO;2-H.

[3] Catalin Cimpanu. *Chrome: 70 percent of all security bugs are memory safety issues*. [Online; accessed 13-August-2020]. 2020. URL: https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/.

[4] M. Dahlweid et al. "VCC: Contract-based modular verification of concurrent C". In: *2009 31st International Conference on Software Engineering - Companion Volume*. May 2009, pp. 429–430. DOI: 10.1109/ICSE-COMPANION.2009.5071046.

[5] Paul Emmerich et al. "The Case for Writing Network Drivers in High-Level Programming Languages". In: *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2019)*. Sept. 2019.

[6] Dawson Engler et al. "Bugs As Deviant Behavior: A General Approach to Inferring Errors in Systems Code". In: *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*. SOSP '01. Banff, Alberta, Canada: ACM, 2001, pp. 57–72. ISBN: 1-58113-389-8. DOI: 10.1145/502034.502041. URL: http://doi.acm.org/10.1145/502034.502041.

[7] Dawson Engler et al. "Checking System Rules Using System-specific, Programmer-written Compiler Extensions". In: *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*. OSDI'00. San Diego, California: USENIX Association, 2000. URL: http://dl.acm.org/citation.cfm?id=1251229.1251230.

[8] David Evans et al. "LCLint: A Tool for Using Specifications to Check Code". In: *SIGSOFT Softw. Eng. Notes* 19.5 (Dec. 1994), pp. 87–96. ISSN: 0163-5948. DOI: 10.1145/195274.195297. URL: http://doi.acm.org/10.1145/195274.195297.

[9] *Fyr compiler source code*. [Online; accessed 13-August-2020]. 2019. URL: https://github.com/vs-ude/fyrlang.

[10] D. Grossman et al. "Cyclone: A type-safe dialect of C". In: *C/C++ Users Journal* 23 (2005/// 2005), pp. 112–139.

[11] Trevor Jim et al. "Cyclone: A Safe Dialect of C". In: *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*. ATEC '02. Berkeley, CA, USA: USENIX Association, 2002, pp. 275–288. ISBN: 1-880446-00-6. URL: http://dl.acm.org/citation.cfm?id=647057.713871.

[12] Richard L. Hudson. *The Journey of Go's Garbage Collector*. 2018. URL: https://blog.golang.org/ismmkeynote.

[13] *Linux kernel in-tree Rust support*. [Online; accessed 13-August-2020]. 2020. URL: https://lkml.org/lkml/2020/7/9/952.

[14] George C. Necula, Scott McPeak, and Westley Weimer. "CCured: Type-safe Retrofitting of Legacy Code". In: *SIGPLAN Not.* 37.1 (Jan. 2002), pp. 128–139. ISSN: 0362-1340. DOI: 10.1145/565816.503286. URL: http://doi.acm.org/10.1145/565816.503286.

[15] *Rust Guide*. [Online; accessed 13-August-2020]. 2020. URL: https://fuchsia.dev/fuchsia-src/development/languages/rust.

[16] *Rust Platform Support*. [Online; accessed 13-August-2020]. 2020. URL: https://forge.rust-lang.org/release/platform-support.html.

[17] Umesh Shankar et al. "Detecting Format String Vulnerabilities with Type Qualifiers". In: *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*. SSYM'01. Washington, D.C.: USENIX Association, 2001. URL: http://dl.acm.org/citation.cfm?id=1251327.1251343.

[18] *Swift versus C gcc fastest programs*. [Online; accessed 13-August-2020]. 2020. URL: https://benchmarksgame-

team.pages.debian.net/benchmarksgame/fastest/swift-gcc.html.

[19] David Tarditi et al. “Checked C: Making C Safe by Extension”. In: *To appear, IEEE Cybersecurity Development Conference 2018 (SecDev)*. Oct. 2018. URL: https://www.microsoft.com/en-us/research/publication/checkedc-making-c-safe-by-extension/.

[20] *The Safe-C Programming Language*. 2018. URL: http://www.safe-c.org.

[21] Torben Weis, Marian Waltereit, and Maximilian Uphoff. “Fyr: A Memory-safe and Thread-safe Systems Programming Language”. In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. SAC ’19. event-place: Limassol, Cyprus. New York, NY, USA: ACM, 2019, pp. 1574–1577. ISBN: 978-1-4503-5933-7. DOI: 10.1145/3297280.3299741. URL: http://doi.acm.org/10.1145/3297280.3299741.