The three main compiler based techniques for full temporal memory safety

Jori Winderickx

KU Leuven

1 Introduction

Low level programming languages are preferred for high performance applications but programmers often struggle to implement the manual memory management. Due to the high performance requirements, memory management is often omitted and programmers need to manually allocate and deallocate the pointers and values in the memory. However, manual memory management can get tedious in larger applications and can result in unexpected behaviour if the memory is not handled correctly. For example, a memory location is freed but the reference is still present and can cause unexpected behaviour because the referenced data is no longer present. The memory errors that can occur are called spatial and temporal memory errors. In higher level programming languages these problems are not common because often memory management is already present. This also causes that they are often slower due to the checks involved for memory management. A lot of research has been done in this area and often solutions that provide partial countermeasures, however with a high performance, are proposed. Countermeasures with full coverage of memory safety also exist and is currently used in implementations, with each having its trade-off in terms of performance, size and usage wise.

A programmer needs to decide which is the right technique for the application. Mostly low level languages are chosen when high performance is necessary or when there are real-time requirements. In these cases the performance can be provided, however memory safety needs to be provided to counter unexpected errors. In this essay the three main techniques that provide full temporal memory safety will be discussed, namely garbage collectors, ownership types and pointer-based checking.

In this essay, first temporal memory errors will be explained. After that the three main techniques to fully prevent these errors will be explored by means of example. Furthermore for each technique the compatibility, with legacy code, and the performance will be discussed to provide a overview of the major advantages and disadvantages.

2 Temporal memory safety

Manual memory management can lead to temporal memory safety violations. These temporal memory safety violations occur when accessing memory locations that have already been deallocated, also called dangling pointers. This means that the pointers after deallocation are still pointing to the locations in the memory that do not contain valid data anymore. As a result these locations can be corrupted, by writing, or falsely interpreted as valid data.

Temporal memory safety is often only an issue for low level programming languages because memory management is not automatically a feature. This is because memory management takes time. The objects in the memory need to be identified, if they are not tracked. Furthermore the objects need to be structured in the memory so that the memory stack can be efficiently used. A way to provide memory management for low level programming languages is to do it manually, allocate and pop the objects from the memory as the program runs. Then problems like temporal memory errors can occur.

In real time applications timing is critical and every event needs to be handled as quick as possible. If the memory management would take too much time to process the memory, the timing may not meet the timing constraints. In this case a high performance technique should be used, however with high performance guarantees often comes other design limitations. For example the memory usage of the memory safety technique could be to large or the program must be written in a different style than the programmer is used to.

The reason that these violations or errors should be countered is that they can cause unpredictable behaviour of programs. For example, a pointer to a date is created and the date is for example used to time an event. If for some reason the pointer would have been deallocated the date itself would be removed from the memory, however without memory safety the pointer to the date could still be used. In this case the date could be falsely interpreted and used to trigger the provided event causing an unexpected behaviour.

These violations can occur in the stack and the heap, examples created by Nagarakatte et al. [1] are shown in Fig. 1. The example on the left causes a violation on the heap. First pointer p is allocated and then pointer q is made a copy of p, both pointers point to the same location on the heap. Then by freeing p, q becomes a dangling pointer since it still points to the freed location on the heap. After the free up of p, the memory pointed to by q can be reallocated by r or any subsequent call to malloc(). The example on the right displays a violation on the stack. The program has a global pointer, this pointer is allocated in the foo() function. The memory allocation of integer a has a local scope and will be popped after foo() returns. This results in the dangling pointer q, pointing to the location of the already popped integer a. At the end of both programs, the memory locations pointed to by q can still be used and therefore causing a temporal memory safety issue.

Heap based

```
int *p, *q, *r;
p = malloc(8);
...
q = p;
...
free(p);
r = malloc(8);
...
... = *q;
```

Stack based

```
int *q;
void foo() {
    int a;
    q = &a;
}
int main() {
    foo();
    ... = *q;
}
```

Fig. 1. Examples of dangling pointer in the heap and the stack

The problem of temporal memory safety are typically only found in low-level programming languages like C and C++. The reason is that these languages provide low-level control of different parts of the system, e.g. manual memory management, hardware control and etc. In higher level programming languages, often a garbage collector is implemented to ensure memory safety. This however results in e.g. slower performance. In the next sections the three main solutions to provide full temporal memory safety will be explored.

3 Garbage collection

Two types of garbage collectors exist: exact and conservative garbage collectors. Exact garbage collectors can only be used in type safe languages because the pointers can be identified at run-time. Not type safe programming languages can use conservative collectors, it treats memory locations as ambiguous references because the pointers cannot be identified at run-time. Programming languages like C and C++ are not type safe because pointers can be directly manipulated whereas managed languages like Java and C#, that are more safely typed, can use the exact collectors [2]. In an ideal case of garbage collectors the definition of garbage is an object that is not needed anymore, however this definition is hard to implement. Conservative garbage collectors therefore use different definitions, e.g. the tracing garbage collector identifies garbage as unreachable objects.

Conservative collectors must ensure correctness by treating the the memory locations as ambiguous references. Meaning that pointers are not known at run-time and the memory is searched for objects. To explain it further, a location in the memory could look like a pointer, however actually be a value, and the memory locations pointed to by these pointers should therefore be reserved. Furthermore the reference could look like data in which case the memory in that location cannot be altered or moved.

3.1 Conservative garbage collectors

Much research has been done in garbage collection, in this section the basic concepts will be explained. For example, the tracing collector, reference counting collector, semi-space collector and etc. The garbage collectors used today are often a combination of these techniques to provide better garbage collectors depending on the situation, e.g. Gen Immix is a copying generational collector [2].

Tracing collector An example of a tracing collector is the mark-sweep Boehm, Demers, Weiser style (BDW) collector [3]. The tracing collectors are based on the definition that garbage is an object that is unreachable. An example of the mark-sweep collector is displayed in Fig. 2. The first step of the mark-sweep filter is the mark stage in which it will trace all active objects in the root set and marks objects as 'in-use' that are referenced by pointers in the root set. In this case objects A and E are first marked and then B and C since they are referenced directly or not directly by the root set. Then in the sweep stage all objects in the memory are scanned and the objects not marked by 'in-use' are freed up. The objects D, F and G are now freed in our example.

The mark and sweep stages result in a temporary halt in the program, stop and collect approach [4], to locate garbage. The advantage is that the collector can find unused and hidden garbage so that the stack and the heap can be cleared. Furthermore searching and clearing the memory of garbage can take time, the speed will be analysed in the Performance section.

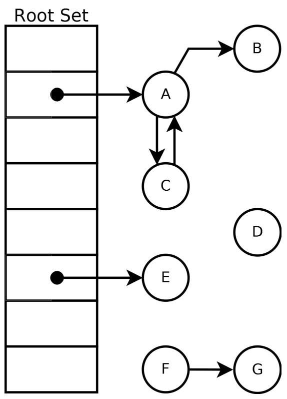


Fig. 2. Basic example of mark-sweep collector

Reference counting collector Another conservative collector is the reference counting collector. Reference counting collectors do not trace the memory stack to find garbage but rely on keeping tab on the amount of references of an object. The definition of garbage is similar to the BDW collector, if no references are left than the object will be unreachable and therefore garbage. When the program changes an object reference, the reference counter can increment or decrement the counter. An example is presented in Fig. 3. In the first situation (a) both pointer p and q point to a distinct integer value(Int a and Int b), both integers have a reference count of one and will not be removed. Then in the second situation (b), due to a modification of pointer p, both pointers reference the integer b. Now integer b has two references and integer a zero, this results in the deletion of integer a on the memory stack.

The advantage of the reference counting is that it continuously keeps track of garbage and clear as the programs deallocates an object. This results in a more predictable collector. Furthermore it does not need to scan the entire memory to find garbage objects. The downside however is that for each object a reference count needs to be stored and maintained. Another problem is the reference-cycle. Reference cycle exist when an object directly or indirectly references itself and the object cannot therefore be removed.

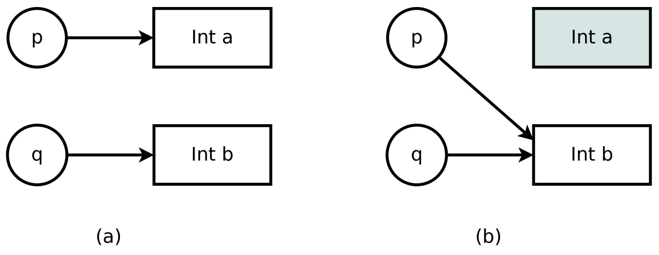


Fig. 3. Basic example of reference counting collector

Semi-space collector Semi space collectors use copying algorithms, meaning that reachable objects are copied to another address during a collection [5]. In a classic semi-space collector the available memory is divided into two equal-size regions, from-space and to-space. During allocation of an object, memory is allocated in the to-space. If then the memory is full, the collection phase starts. During a collection, the roles of the regions are swapped and then only live objects are copied from the from-space into the to-space. At this moment all the references of the live objects need to be updated. This is done by storing the forwarding-addresses of the moved objects during the collection phase.

The advantage of the copying collector to the mark-sweep collector is that there is no need to keep the list of free memory and fragmentation is avoided. This is because during collection phase the memory is restructured via the copying of only live objects. However it requires more memory because half of the memory cannot be used. Furthermore depending on the amount of live objects the copying collector can be more efficient than the mark-sweep collector. If only a few objects are alive, the collector does not need to copy as much objects while the mark-sweep collector always analyses the entire memory.

The semi-space collector has since its introduction been improved upon, for example the mostly copying collector introduced by Bartlett [6]. The design of the mostly copying collector has since its introduction remained popular [2]. The Bartlett's collector differs from the classic semi-space collector in two ways. Firstly, the to- and from-space are logical spaces comprising of linked lists of pages on the memory. And secondly, at the start of each collection, pages referenced by an ambiguous root are promoted. This results in adding the promoted pages into the linked list of the to-space. The promoted pages are then used as the roots for a final copying collection phase, in the Bartlett collector it is assumed that all objects on promoted pages are alive.

Generational collection Generational collection is a basic concept that is based on the assumption that recent created objects are more likely to be garbage than objects that have been alive for a longer time. In this case the heap is often divided in multiple generations. If a object is not directly cleared, it will be promoted into the next generation. Older generations are less often checked for garbage because of the assumption that recent created objects are more likely to be garbage.

3.2 Exact garbage collectors

In type safe programming languages, the type of allocated objects are known and cannot be misinterpreted. At allocation the type is identified and during execution the run-time must dynamically identify root references and free not used resources. The basic concepts of exact collectors are often similar to the previously explained concepts of conservative collectors.

3.3 Compatibility

Using conservative garbage collector in a new c program ensures that the programmer does not need to think about memory management a lot. Some consideration should be taken so that the garbage collector can work efficiently. For legacy code using manual memory management, some adaptation should be considered e.g. removing the manual memory management.

3.4 Performance

The work by Shahriyar et al. [2] analyses the performance of conservative garbage collectors. The performance benchmarks where done on a Ubuntu 12.04.3 LTS server distribution of a 64-bit Linux and the hardware contained a 3.4 GHz, 22nm Core i7-4770 Haswell processor and 8 GB of DDR3-1066 memory. In a full process of a garbage collector, the collection of garbage and the restructure of the memory, can take for the BDW collector 255 milliseconds up to 15668 milliseconds, depending on the benchmark, with a mean of 1956 milliseconds. The reference counter based conservative collector is a bit faster, with times ranging from 208 milliseconds to 12512 milliseconds with a mean of 1729 milliseconds. This proves that in case of severe real-time constraints a garbage collector can cause too long lock-ups because the garbage collector will freeze the program to provide memory management.

The differences of conservative garbage collectors to their exact variants are very small. Sharhriyar et al. [2] analysed this difference, the result is that all the conservative collectors are a bit slower, however with very little overhead ranging from 1% for the BDW collector to 9.3% for the most copying collector.

4 Ownership types

To ensure temporal memory safety, ownership type memory management defines that an object can never refer to another object which has a possible shorter lifetime. It can be seen as enforcing a topology on the patterns of references [7]. In this chapter the Rust programming language [8] will be used to explain ownership types. Rust uses a ownership type concept for memory safety while preserving performance. The concept of Rust has three rules: lifetime, ownership and borrowing.

Ownership The first rule we will explore is ownership. The basic idea behind this rule is that variables in the memory are owned through variable bindings. A variable binding has the property that they have ownership of what they are bound to, meaning when the binding is no longer valid the owned memory locations will be freed. A binding can become no longer valid when e.g. they go out of scope. A first example is shown in Fig. 4. At the start of the function foo(), vector v is initialised on the stack and it's values on the heap. Then at the end of foo(), vector v goes out of its scope and will therefore expire. Since the vector $[1, 2, 3]$ is owned by v, the reference v and the vector will be popped from the memory.

```
fn foo() {
    let v = vec![1, 2, 3];
}
```

Fig. 4. Lifetime of variables in Rust, the scope of variable v is the function foo(). At the end of foo() the variable v and anything related to it will be cleaned up.

A resource is always bound to exactly one binding, however the resource can be re-assigned. In the Fig. 5, the move semantics is displayed. During initialisation the vector is assigned to v after which the resource is re-assigned to v2. At that moment the reference v cannot be used since it does not have ownership of the vector anymore. Another semantic that can occur, depending on the type of the resource, is called copy. It is as clear as the name suggest, types like integers can be entirely copied and as a result keeping the original reference and variable binding. For example, if in the example of Fig. 5 a integer was created the copy semantic would have been used instead of the move semantic. In this case a new distinct integer v2 would have been created while also preserving the integer v.

```
let v = vec![1, 2, 3];

let v2 = v;
```

Fig. 5. Move semantics in Rust, ownership of the vector $[1, 2, 3]$ is passed upon v2 and the v reference is not assigned and therefore removed.

In case of functions, moving the ownership can get very tedious as visualised in Fig. 6. Because the vectors v1 and v2 are handed to the function as input, the ownership is also moved to that function. If the function would then end without handing the ownership back, the vectors would go out of scope and be freed in the memory. In this example the ownership of the vectors are handed back via the return of the function.

```
fn foo(v1: Vec<i32>, v2: Vec<i32>) -> (Vec<i32>, Vec<i32>, i32) {
    // do stuff with v1 and v2

    // hand back ownership, and the result of our function
    (v1, v2, 42)
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let (v1, v2, answer) = foo(v1, v2);
```

Fig. 6. Ownership of variables in functions programmed in Rust. After a function the ownership needs to be handed back and in this case the function will return the values as a result of the function.

These rules ensure that only one reference can be used to alter the referenced data. This protects against e.g. race conditions. Race condition occur where two independent and parallel running functions try to alter a resource at the same time. This can result in unexpected value of the resource.

Borrowing A resource can be borrowed instead of moved to preserve the original reference. Using the ($\&T$) reference type, the program does not pass the resource however it passes the reference. This is displayed in Fig. 7. In this example both vectors are 'borrowed' to the function foo(). The function foo() can then use the vectors in its calculations, note that by using $\&T$ type an immutable reference is passed. The foo() function will not be able to change the values of v1 and v2.

```
fn foo(v1: &Vec<i32>, v2: &Vec<i32>) -> i32 {
    // do stuff with v1 and v2

    // return the answer
    42
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let answer = foo(&v1, &v2);

// we can use v1 and v2 here!
```

Fig. 7. Borrowing a variable in Rust via the immutable reference type.

In Rust two types of references exist: immutable ($\&T$) and mutable ($mut\ T$) references. The rules for these types of references are that a resource can have mutiple immutable references but a resource can only have one mutable reference. This is done to prevent data races, as mentioned before, because now only one reference can be used to write to a resource and multiple references can be used to read the resource. An example of the mutable reference is presented in Fig. 8. The mutable reference is borrowed for y, therefore y can change the resource first pointed to by x. After the curly braces the scope of reference y ends and the borrow and reference y will therefore be freed. Now x has again full control of the resource.

```
let mut x = 5;

{
    let y = &mut x; // -+ &mut borrow starts here
    *y += 1;        //  |
}                   // -+ ... and ends here

println!("{}", x);  // <- try to borrow x here
```

Fig. 8. Borrowing a variable in Rust via the mutable reference type.

Another memory issue prevented by scope and reference in Rust is the 'use after free'. As displayed in Fig. 9, in the scope of the brackets the y reference is replaced with the reference of x. Reference y is now pointing to resource of x, however after the scope ends (closing bracket), x will be freed causing the y reference to look like an dangling pointer. In Rust, further use of the y reference will result in an error.

```
let y: &i32;
{
    let x = 5;
    y = &x;
}

println!("{}", y);
```

Fig. 9. Using the resource after freeing it up from memory will result in an error.

Lifetime The concept of Lifetimes in the ownership model of Rust is complementary to the other concepts to counter errors like the dangling pointer. For example, if an object is borrowed to a function and this function decides to deallocate the reference, the reference would be perceived to be still valid for the original owner, causing a dangling pointer. To counter this, references are accompanied with lifetimes. This can be done through implicit or explicit lifetime usage, displayed in Fig. 10. In rust $'a$ represents the lifetime of a reference. In the explicit example, the lifetime $'a$ is passed via the general parameters of a function, via the $< .. >$ input. Then this lifetime is allocated to the x reference. If the lifetime is not explicitly added, the compiler will add lifetimes to the objects according to the lifetime elision rules of Rust.

```
// implicit
fn foo(x: &i32) {
}

// explicit
fn bar<'a>(x: &'a i32) {
}
```

Fig. 10. Explicit and implicit defining the lifetime of objects

An example on the duration of lifetimes is given in Fig. 11. The lifetime of an object in Rust is often correlated to the scope in which the variable is valid. In this example, the reference y and struct f are initialised in the beginning of the main function. Both variables have the scope of the main function in which they are valid and this can also be visualised as the lifetime of the variables.

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let y = &5;           // -+ y goes into scope
    let f = Foo { x: y }; // -+ f goes into scope
    // stuff              //  |
                          //  |
}                         // -+ f and y go out of scope
```

Fig. 11. An example on the duration of lifetimes

The code in Fig. 12 provides an example on how Rust can prevent dangling pointers. In the beginning of the main function the variable x is initialised. Then in another scope, reference y and struct f are initialised. At that moment the value of x is set as the pointer to the reference x of the struct f. The problem is that the scope of struct f will end at the bracket whereas the lifetime of variable x is longer and will only end on the closing bracket of the main function. Since the lifetimes of x and f.x do not match, Rust will emit an error.

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let x;                    // -+ x goes into scope
                              //  |
    {                         //  |
        let y = &5;           // ---+ y goes into scope
        let f = Foo { x: y }; // ---+ f goes into scope
        x = &f.x;             //  | | error here
    }                         // ---+ f and y go out of scope
                              //  |
    println!("{}", x);        //  |
}                             // -+ x goes out of scope
```

Fig. 12. An example on how Rust would prevent the dangling pointer

Languages like Rust with ownership types have strict rules that may inconvenient the programmer to write the application. However, by using these rules Rust can provide memory safety while also limiting the overhead of ensuring memory safety.

4.1 Compatibility

If a programmer needs to follow the rules if he wants to use ownership type memory management, e.g. using Rust requires the programmer to learn this new language in order to write a program. The same applies to legacy code, it will need adaptation to another compiler or language depending on the selected technique.

4.2 Performance

The performance of Rust in terms of pointer usage is very similar to C, as evaluated by Lin et al. [9]. They compared the implementation of a high performance garbage collector programmed in C and Rust. The benchmark is based upon the time spent on allocating, marking and tracing 50 million objects, results are presented in Table. 1. The benchmark were run on a computer with Linux kernel version 3.17.0 with a $22nm$ Intel Core i7 4770 processor. Rust proves for this application to have a small overhead in terms of performance, approximately $4ms$ longer execution time in case of the alloc and trace benchmark, for the high performance garbage collector.

Table 1. Average execution time of performance critical paths in high performance garbage collector programmed in C and Rust.

	C	**Rust (% to C)**
alloc	$370 \pm 0.1ms$	$374 \pm 2.9ms(101\%)$
mark	$63.7 \pm 0.5ms$	$64.0 \pm 0.7ms(100\%)$
trace	$267 \pm 2.1ms$	$270 \pm 1.0ms(101\%)$

5 Pointer-based checking

Via fat pointers, pointer-based checking can ensure temporal memory safety. In this case metadata is stored along with every pointer created, displayed in Fig. 13. The metadata provides the pointer, information to provide temporal and spatial memory safety e.g. boundaries of the value and etc. In this example the metadata is stored together with the pointer, however this approach does require that existing code's manual memory management to be rewritten since a fat pointer utilises more memory. Using fat pointers will therefore result in a different memory layout. Additionally, unsafe type casts of the pointer can result in corruption of the metadata. These problems has been solved by disjoint metadata storage, for example storage in a Trie structure in the CETS implementation created by Nagarakatte et al. [1]. Another solution is to use shadow memory, implemented by Nagarakatte et al. [10]. In this type of metadata storage, a distinct memory is used to store the metadata of the pointers. The disadvantage of disjoint metatdata storage to fat pointers is that the accesses(checking the metadata) are slower.

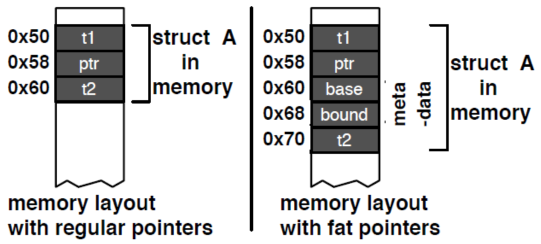
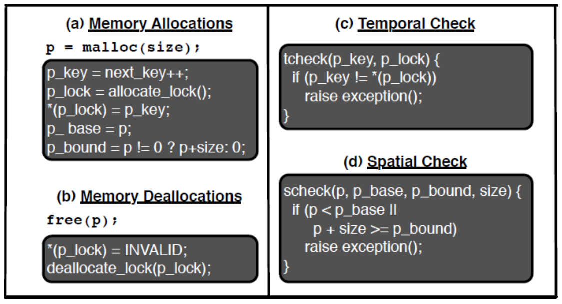


Fig. 13. Memory layout differences of regular and fat pointer

Using SoftBoundCETS as example, each pointer is paired with two objects, the key and the lock. They key is an allocation identifier and the lock is a reference to a location in the memory called the lock location. If the key and the value in the lock location are the same then the pointer is still valid. The key and lock are used so that a validation check of the pointer is a simple load and comparison of the key and lock location. Temporal safety is thus ensured by allocation and checking of the metadata, pictured in Fig 14. On allocation of a pointer, metadata is created(Fig. 14a) to fit the description of the allocated memory, for temporal safety a key, lock reference and the lock location objects are created. If a pointer is then freed, the metadata will be invalidated and locked by changing the value of the key or lock location(Fig. 14b). After invalidation the pointer will not be usable. If the pointer is addressed a validation check is performed by reading and comparing the key value and the lock location value(Fig. 14c). Spatial checks are also present and consist of boundary checking, however this will not be discussed further.

Metadata storage The metadata consist of base and bound object for spatial memory safety and the key and lock object for temporal memory safety, pictured in Fig. 15. Furthermore in SoftBoundCETS a shadow space is used to store the pointers. This is done to prevent corruption and leaves the memory layout intact. In the example, there are two pointers p and q each pointing to a distinct location. With each pointer metadata is



```
p = malloc(size);
p_key = next_key++;
p_lock = allocate_lock();
*(p_lock) = p_key;
p_ base = p;
p_bound = p != 0 ? p+size: 0;
```

```
free(p);
*(p_lock) = INVALID;
deallocate_lock(p_lock);
```

```
tcheck(p_key, p_lock) {
  if (p_key != *(p_lock))
    raise exception();
}
```

```
scheck(p, p_base, p_bound, size) {
  if (p < p_base II
      p + size >= p_bound)
    raise exception();
}
```

Fig. 14. How pointer-based checking can ensure temporal memory safety. (a) Creation of a fat pointer, (b) metadata invalidated on deallocation, (c) lock and key checking, and (d) bounds spatial check

stored in the shadow memory. The difference between the two metadata objects is only the spatial objects, base and bound object, because the location and boundary is different for each pointer. The temporal objects are however the same since both locks references the same lock location and both pointers are valid.

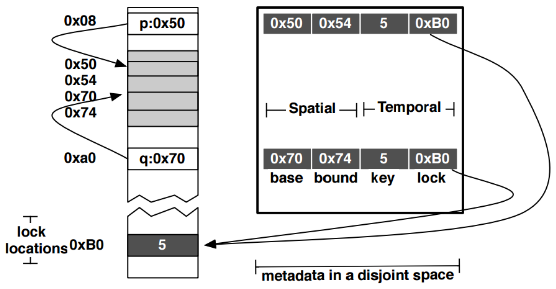


Fig. 15. Storage of metadata for each pointer in shadow memory of SoftBoundCETS

Disjoint storage To store the metadata in a disjoint space a couple of considerations need to be taken into account, how are the metadata locations mapped to the memory locations, is metadata provided for all memory locations and when is the metadata updated. For the SoftBound-CETS project, it was chosen to use a two level trie data structure that maps the entire virtual address space, the entries in the trie are only allocated on creation of the metadata. In this project the metadata is only stored for pointers in the memory and performs metadata loads only when value written to the memory contains a pointer.

Pointer-based checking techniques e.g., SoftBoundCETS project, only perform the memory safety checks to provide more reliable programs. The programmer still needs to manually manage the memory. Furthermore the advantage of the SoftBoundCETS project is that the programmer does not need to alter his program because the memory layout will not change.

5.1 Compatibility

A program in C with pointer-based checking does not require additional adaptation if the SoftBoundCETS project is used and therefore providing high compatibility. For example, legacy code with manual memory management does not require change of the code, it only requires the developer to use a different compiler so that the pointer based checking features are used.

5.2 Performance

A performance evaluation of the SoftBoundCETS instrumentation is done by Nagarakatte et al. [11]. The evaluation is based on benchmarks e.g., lbm, go, equake and etc., to analyse the overhead of the full checking i.e., enforcing both spatial and temporal safety, and the overhead of temporal and spatial safety checking separately. The average overhead caused by the pointer-based checking is 108% or in other words the execution time with full checking takes the program 2.08 times longer. For providing only temporal safety the average overhead is approximately 40% or 1.4 times longer execution.

6 Conclusion

In the previous chapters temporal memory safety is explored via the three main techniques, garbage collection, ownership type and pointer based checking. All of the three techniques have their own trade-offs resulting in that the usage will depend on the application requirements. In the first case of garbage collection all memory management can be controlled via the garbage collector and the programmer does not need to worry about memory management. The downside of garbage collectors is the large performance overhead caused by checking and managing the memory and the lock-up it provides, therefore garbage collection is not very usable in real time applications. The major advantage is the abstraction it provides to memory management because manual memory management is no longer needed. The second technique, Ownership types is explained via the language Rust. The major advantage of ownership types is its low performance overhead, the downside however is that it requires the programmer to follow some programming rules, in the Rust language: scope, borrowing and lifetime rules. The last explained countermeasure to temporal errors is pointer-based checking. It provides checks to counter temporal errors but does require the programmer to manage the memory as originally. Depending on the implementation used, the programmer also may need to take the fat pointers into account which can change the memory layout. Furthermore the overhead caused with full memory safety is a bit larger than the ownership type performance but is lower and more predictable than the garbage collector approach.

References

1. S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "CETS: Compiler-Enforced Temporal Safety for C," *Proceedings of the 2010 international symposium on Memory management - ISMM '10*, p. 31, 2010.
2. R. Shahriyar, S. M. Blackburn, and K. S. McKinley, "Fast conservative garbage collection," *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications - OOPSLA '14*, pp. 121–139, 2014.
3. A. Demers, M. Weiser, B. Hayes, H. Boehm, D. G. Bobrow, and S. Shenker, "Combining Generational and Conservative Garbage Collection: Framework and Implementations," *17th Annual ACM Symposium on Principles of Programming Languages*, pp. 261–269, 1990.
4. H.-J. Boehm and M. Weiser, "Garbage collection in an uncooperative environment," *Software: Practice and Experience*, vol. 18, pp. 807–820, sep 1988.
5. R. R. Fenichel and J. C. Yochelson, "A LISP garbage-collector for virtual-memory computer systems," *Communications of the ACM*, vol. 12, pp. 611–612, nov 1969.
6. J. F. Bartlett, "Compacting Garbage Collection with Ambiguous Roots," *ACM SIGPLAN Lisp Pointers*, vol. 1, no. 6, pp. 3–12, 1988.
7. T. Zhao, J. Baker, J. Hunt, J. Noble, and J. Vitek, "Implicit ownership types for memory management," *Science of Computer Programming*, vol. 71, no. 3, pp. 213–241, 2008.
8. N. D. Matsakis and F. S. Klock, "The rust language," *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology - HILT '14*, vol. 34, no. 3, pp. 103–104, 2014.
9. Y. Lin, S. M. Blackburn, A. L. Hosking, and M. Norrish, "Rust as a language for high performance GC implementation," in *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management - ISMM 2016*, (New York, New York, USA), pp. 89–98, ACM Press, 2016.
10. S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Everything You Want to Know About Pointer-Based Checking," *LIPIcs - Leibniz International Proceedings in Informatics*, vol. 32, p. 208, 2015.
11. S. G. Nagarakatte, *Practical Low-overhead Enforcement of Memory Safety for C Programs.* PhD thesis, University of Pennsylvania, 2012.