DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

UFMG

Projeto Orientado em Computação II (DCC009)

Proposta Mista

Liveness Aware Memory Allocator for Honey Potion

Kael Soares Augusto

Advisor: Fernando Magno Quintão Pereira

Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Belo Horizonte, MG, Brazil
December 07, 2025

Abstract. Honey Potion is a project developed by the Laboratory of Compilers of UFMG with the objective of making eBPF technology accessible. However, dealing with eBPF brings a severe limitation: 512 bytes of stack. This project proposes and implements a smart allocator that is liveness aware to circumvent that limitation. As a result, memory usage has reduced to 44.6% on average and *over 5%* in the best scenarios when compared to the original implementation across 24 benchmarks.

1. Introduction

Honey Potion is a framework that translates Elixir code to eBPF. It is fully transparent and provides intermediary files such as the .bpf.c files that eventually become eBPF bytecodes. eBPF allows users to run bytecode in the Linux kernel within a sandbox that gives the user the power to check and modify system calls, package and more without requiring the use of kernel modules or other invasive solutions [Vie+20].

However, to keep the kernel safe, eBPF goes through the eBPF verifier, which guarantees that: the program doesn't have loops, verifiably terminates, has up to one million instructions, guaranteed safe memory accesses and, most importantly, 512 bytes of stack [Vis+23].

The limitations of the eBPF verifies introduces many challenges to Honey Potion. As an example, consider the following excerpt from one of the benchmark programs:

```
def main(_ctx) do
  # To get elements from the map, use bpf_map_lookup_elem from Bpf_helpers!
  entry_0 = Honey.BpfHelpers.bpf_map_lookup_elem(:Example_map, 0)
  entry_1 = Honey.BpfHelpers.bpf_map_lookup_elem(:Example_map, 1)
  entry_2 = Honey.BpfHelpers.bpf_map_lookup_elem(:Example_map, 2)
  entry_3 = Honey.BpfHelpers.bpf_map_lookup_elem(:Example_map, 3)
  entry_4 = Honey.BpfHelpers.bpf_map_lookup_elem(:Example_map, 4)
  entry_5 = Honey.BpfHelpers.bpf_map_lookup_elem(:Example_map, 5)
  entry_6 = Honey.BpfHelpers.bpf_map_lookup_elem(:Example_map, 6)
  entry_7 = Honey.BpfHelpers.bpf_map_lookup_elem(:Second_Example_map, 0)
  #entry_8 = Honey.BpfHelpers.bpf_map_lookup_elem(:Example_map, 8) Removing this makes it go over memory

  # To update elements from the map, use bpf_map_update_elem from Bpf_helpers!
  Honey.BpfHelpers.bpf_map_update_elem(:Example_map, 0, entry_0 + 1)
  Honey.BpfHelpers.bpf_map_update_elem(:Example_map, 1, entry_1 + 2)
  Honey.BpfHelpers.bpf_map_update_elem(:Example_map, 2, entry_2 + 3)
  Honey.BpfHelpers.bpf_map_update_elem(:Example_map, 3, entry_3 + 4)
  Honey.BpfHelpers.bpf_map_update_elem(:Example_map, 4, entry_4 + 5)
  Honey.BpfHelpers.bpf_map_update_elem(:Example_map, 5, entry_5 + 6)
  Honey.BpfHelpers.bpf_map_update_elem(:Example_map, 6, entry_6 + 7)
  Honey.BpfHelpers.bpf_map_update_elem(:Second_Example_map, 0, entry_7 + 8)
  #Honey.BpfHelpers.bpf_map_update_elem(:Example_map, 8, entry_8 + 9) Removing this makes it go over memory
```

Figure 1: Honey_Maps.ex's function.

The 512 byte stack limit means that that programs isn't considered valid when removing the last two comments. This shows that the limit already impacts relatively small programs.

Although it is not very popular, eBPF is used by big companies such as Google, Netflix and Cloudflare for network security, packet processing and performance monitoring [eBP]. The objective of Honey Potion is to construct a user-accessible compilation pipeline from Elixir to eBPF to bridge the gap from the low accessibility to the powerful features of eBPF.

The solution implemented to resolve the 512 byte stack limit comes in two parts:

1. Moving variables from the stack to eBPF Maps

2. Making the allocator liveness aware.

eBPF maps are data structures that share memory between the user-space and the kernel-space in a way that doesn't limit eBPF's program size. As a result, we can create an impromptu heap on it. This by itself already solves the problem of the 512 byte limit. However, much more could be done to improve the memory consumption of the program.

Making the allocator liveness aware means that we are capable of re-using the memory space occupied by variables that have no further uses (considered dead) for new variables. Consider the following simple example:

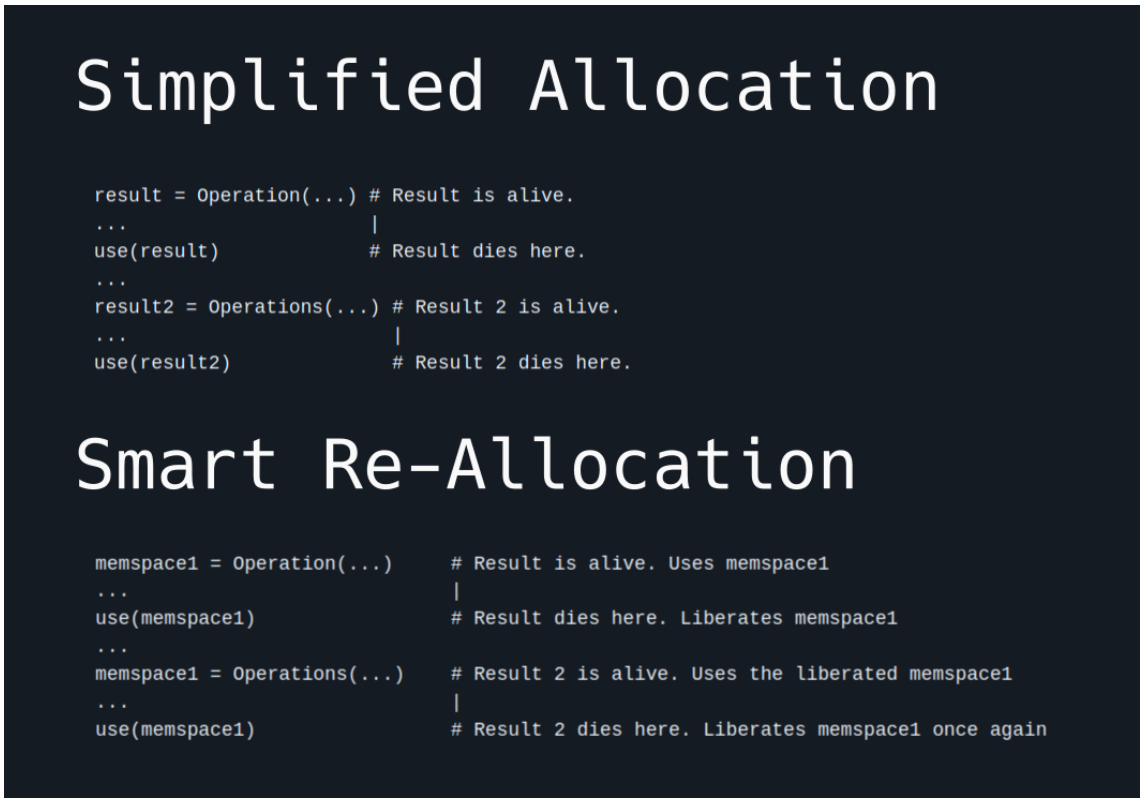

Figure 2: An example of smart reallocation.

From now on, liveness aware memory allocator and smart allocator will be used interchangeably. This paper will be focusing on the second part of the solution, as it is the most academically relevant and has significant metrics, such as on average approximately 2x improvement in memory consumption.

The next section will talk about the context of the project, the general solution idea and other references used to gather the information for the project (Section 2). After that, the constributions and implementation will be detailed on Section 3. Finally, results and possible future work will be given in Section 4.

2. Context and References

Translating Elixir to eBPF verifier-ready C code is a significant challenge, given the differences between the languages. Consider the figure below as a reference to the paradigm comparison between the source and target languages:

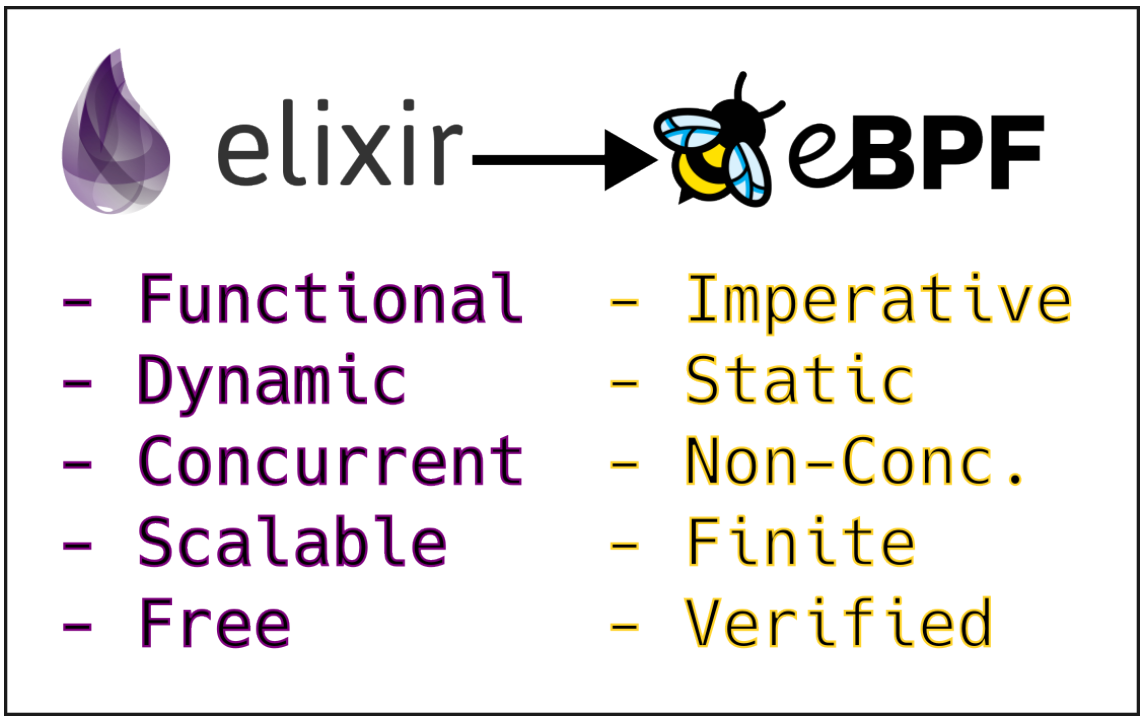

Figure 3: Comparison between the paradigms of Elixir and eBPF

Honey Potion has many of the steps of a compiler, however it starts from the Elixir AST instead of from a programming language. Honey Potion is a meta-compiler [MD15], that means that we functionally skip the front-end of the compiler and go straight to the back-end. The objective of our compiler isn't to reach binary, but instead to reach a verifier-ready C code. More details on this process, what is verifier-ready code and how Honey Potion works can be found in the Honey Potion Paper[Aug+25].

The Elixir AST is represented as a sequence of 3-tuples with the format `{:operation/atom, metadata, arguments}`. Code blocks are represented by `:block` and conditionals as `:case` or `:cond`. Honey Potion grabs the AST of the Elixir program and feeds it through optimizations and eventually into the translator.

Elixir manages it's own memory without need for the user to allocate or deallocate objects. However, C does not manage memory allocations for the user, requiring direct calls to malloc and free as an example. Transferring from one behavior to the other brings in some challenges that are limited by the finite imperative style imposed by eBPF.

To counter the limited size of eBPF's stack, we use a feature called eBPF maps. These are data structures that interface with both the kernel-side and user-side, allowing for communication with both sides and the storage of bigger amounts of data. Arbitrarily, 4096 bytes were allocated into the eBPF map to serve as the new stack. As needs arise, that number can be made larger. However, it seems unlikely given variable reuse.

From now on, the eBPF map allocated for the stack will be called map_stack. It is consisted of an `BPF_MAP_TYPE_PERCPU_ARRAY` (pretty much an array) with one entry of type `char[4096]`. To get and set attributes into the map_stack we have to use type casting and pointer de-referencing to change from the single-byte char to the multi bytes of `int`, `String` and `Generic`.

Given that the types mentioned have different sizes, we must be aware of which positions are free on the byte level, taking care to manage the correct size for each variable and position. This becomes even more important when considering the reusing of free space, which requires managed allocation and deallocation.

To manage free memory we use a data structure called *memory blocks*. More specifically, we use a linked list of tuples, each with a free position and it's size. When allocating, we look for the first tuple that has enough size, removing that size from the total and adding it to the start position. When giving back memory, we return it to it's proper position, annexing it to adjacent blocks. As a result, we have an ordered list of positions that we can fill and their sizes.

To check what variables can be deallocated we use liveness analysis. Liveness analysis is a type of static analysis (runs before compilation) that checks for every program point what variables are alive, as in, has been declared and will be used and which are dead, as in, has been declared, but won't have any more uses.

Sometimes there will be free blocks, but none big enough to fit a new variable. For times like those we can defragments our memory: We move all variables to the left-most position available, accumulating all gaps on the right. As a result, we change the state of the stack_map to fit more elements by summing up the free sizes on the right-most position.

However, we cannot change the state of the stack_map during a branch, because as it merges back into the code we may have conflicting states from each of the branches. Given the limited size of eBPF programs, if more flexibility is needed, a tree-splitting algorithm can be used, effectively splitting branches by duplicating every tree after a branch to avoid merges, as can be see in Figure 4 in Section 3.1.

Originally, the eBPF translator would simply declare every variable on the Elixir side into the eBPF side when traversing the AST. This method will be referenced when comparing to the new smart allocator. Our new smart allocator now liveness aware and is capable of reducing memory usage thanks to all of the structures described above.

Defragmentation was not required in any of the programs and thus won't be discussed in Section 4. However it will be mentioned in Section 3 as it was implemented.

More about Honey Potion can be found in Dwctor's youtube channel [Dwc] and in Honey Potion's github repository [Lab].

3. Contributions

The current implementation of the Smart Allocator has two prerequisites: **Liveness Analysis** and **no merges** in the AST. The first is required to know what variables are alive or dead to deallocate them; this was previously implemented by the author and thus will not be discussed further. The second, named **Tree-Split** is required to allow a single truth for the state of memory in each branch. This way, we delegate the memory to a compile-time designation.

With the two prerequisites fulfilled the main implementation is done. This implementation consists of a Greedy Memory Allocator that has defragmentation and will be called **Smart Allocator**.

3.1. Tree-Split

To better understand Tree-Split's effect in an AST, consider the following example:

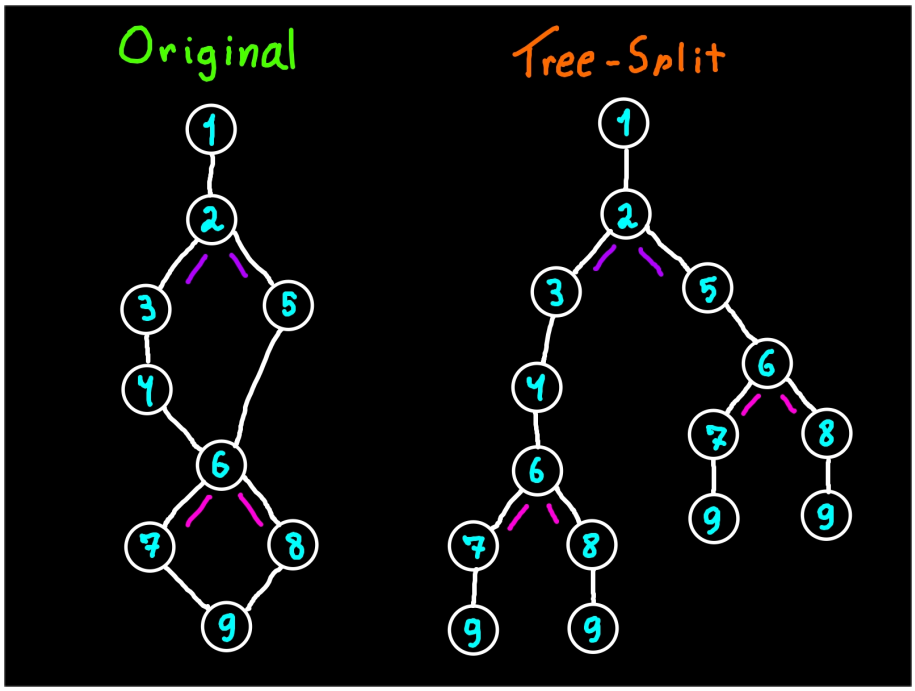

Figure 4: A program before and after being tree-split.

Figure 4 shows an example where branches 2 and 6 are never merged by copying everything below 6 and 9 to where the branches would have merged. This now gives us a program representation where any possible branch of code is a sequential line of instructions. As a result, we need not worry about any other branches when optimizing.

In our case, this is most relevant because we don't need to think about phi functions, which are responsible for merging optimizations across different branches. This decision was taken because standardizing a join of two memory states could be both complicated and costly during execution time.

3.2. Smart Allocator

By default, the Smart Allocator is greedy: it looks for the first memory space that a new variable fits in and places it there. When these variables are considered dead (variable

analysis) we deallocate them. Variables that are created by the Honey Potion translator called *helper_variables* are always scope-bound to the functions that created them, so they are allocated and freed as soon as possible.

In case we have to add a new variable, but there are no spaces big enough to fit it, we defragment the memory ”to the left”, freeing up the sum of the free blocks to the right. If it still doesn't fit, then we do not have enough memory for this program and terminate translation with a warning. Tree splitting is a necessity so that we don't have to deal with merging different memory states after defragmentation differences at run-time. This has been futher detailed in Section 2.

3.3. Implementation

This section is a check-list of implemented features and modules.

3.3.1. Context

One of the main changes done in this project was adding and maintaining a live ”context” throughout compilation, that keeps in mind the map_stack state. This is done through 3 data structures:

- Map %{var_name -> {pos, size}}
- Map %{pos -> {var_name, size}}
- Free Memory Blocks

The first one is responsible for keeping track of where are the variables so that they can be translated, the second one is responsible for helping the defragmentation by giving us a list of positions that can be manipulated and the third one keeps track of where we can allocate new variables.

3.3.2. Memory Blocks

To represent the free memory blocks, a new module called `MemoryBlocks` was created in the runtime folder of the project. This module is considered runtime as it is used and modified throughout the ongoing translation process. As mentioned before, it keeps a list of contiguous free memory blocks with the positions and sizes in ordered fashion. To upkeep this structure, a few methods were devised:

get(memory, requested_size). Goes through the available memory blocks seeking one with size equal or higher than the requested size. If is is found, the memory is updated to lack that space, by reducing the memory block that it was taken from, or removing it if it's size becomes 0. If it is not found, the position returned is -1.

give(memory, pos, size). Seeks through the memory blocks until a position where the pos is in-between the two positions before and after it. We then add in that position the size that is returned. However, we must be aware that what we added might be contiguous with the blocks before and after, which may lead to a fusing of elements in the process.

defrag(memory). Sums up the sizes of all of the free blocks and adds them to the last one, subtracting from it the size that was spent on the others, resulting in only one big block at the last starting position available.

3.3.3. Translator Methods

More methods than the ones listed here were created. The main ones and their functionalities are listed below:

get_code_value. Gets the name of a code block (returns a variable) in a given context within the stack. The string returned is the C representation of the value returned by that code block, represented as `(*(<type>*) (stack + <var_pos>))`. Given that the position may change thanks to defragmentations, we need the context to keep track of the current and future positions.

allocate_var. Grabs a position from the free memory blocks, returning the variable and an up-to-date context with the updated memory blocks, variable to position and size map and position to variable and size map. In case there isn't any available space, it calls defragmentation. If again there isn't enough space an error is raised for lack of stack. Also, if the program didn't run Tree Split as described in Section 3.1 the first defragmentation should immediately raise an error.

deallocate_code. Checks if the variable is alive. If it's not alive, or is a helper_variable, we deallocate it by updating the context structure to return the memory block and drop the variable name and keys from the context. The translator should call this function every time a variable that it requested through translation has been used.

defragment_context. Uses the position to variable name and size map to move all variable positions to the first one available one by one. For every one of them, create a piece of code that moves the code from the old to the new position, always from left to right so that no overwriting is done. We then return the context and the code with the updated positions and the necessary code for reallocation to make them valid.

3.3.4. Changes to to_c translation

The `to_c` function is responsible for translating the Elixir AST into the `TranslatedCode` structure, which contains code, a returned variable, the type of that variable and the context. Previously `to_c` used to have as input the AST element to be translated and a possible context that contained only the eBPF maps of the program.

The first step was to adapt all of the `to_c` functions to take in both an AST and a context always and to return the context back to the caller so that variables created in one context can be used in the one above by accessing the context with the name and the functions described above.

Next step was changing all variable declarations to allocations into the stack, with the defragmentation code previous to the declarations, and deallocating variables that you gathered from the calls that you made as you are the one that requested them. If they are an alive variable, they are kept, otherwise they are removed.

Finally, all helper functions in the `translator.ex` had to be changed in similar ways to accommodate the new paradigm.

3.3.5. Boilerplates

New boilerplate was made to support the new runtime structures used with the new memory allocation and old boilerplate that became deprecated was removed. Honey Potion already had sections for generating boilerplate code, so a few changes were done straight to it: First we defined the stack_map, the eBPF Map where we keep the stack, in the `runtime_structures.bpf.h` file included in every eBPF program. Variables to get the stack_map, turn it into a pointer and access it with 4, 8 and 12 bytes of size can be found in `boilerplates.ex`. As a result, instead of using up many of the 512 bytes we had with our programs, we use 6 pointers: `Stack` to get the stack_map and check that is was returned, `stack` to convert it into a char*, `stack_int; stack_str; stack_str;` to access 4, 8 and 12-byte sized elements and `lookup` to get values from eBPF maps and return them if they exist.

3.3.6. Misc.

Tree-Split. As mentioned in Section 3.1, Tree-Splitting was implemented using the AST of the user program given to us by Elixir.

Liveness & Type Analysis. Honey Potion has a liveness and type analysis. This allows us to both know when we can deallocate variables and what are the sizes that they require when being allocated.

Type Propagation. A few extra types had to be added and accounted for during translation for their new representation in memory.

4. Conclusions

The main objective of this project was improving memory consumption in Honey Potion. Thus, we decided on answering two research questions:

- **RQ1**: How much does Tree-Split worsen the size of compiled binaries?
- **RQ2**: What are the gains in memory consumption with the smart allocator?

4.1. RQ1 Tree-Split's effect on binary size

Surprisingly Tree-Split had minimal effect on program size: almost all of them kept the same exact size as shown in Figure 5.

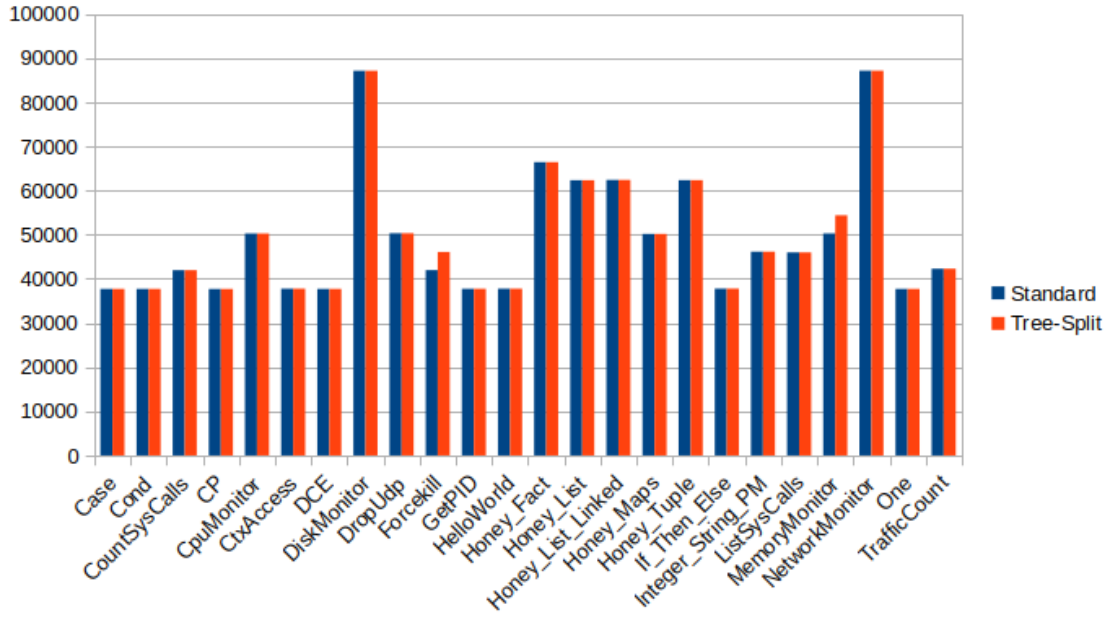

Figure 5: Absolute memory consumption in bytes.

Even though many other programs also had branches, the only ones with size differences were Forcekill and NetworkMonitor both with 4096 bytes more.

As a result, it is reasonable to conclude that tree-split did not have significant changes in binary size.

4.2. RQ2 Smart-Alloc memory gains

The new Smart Allocator will be compared to the original allocator already present in Honey Potion. This will be done in an overlay graph: the results of the Smart Allocator will be placed in front of the original for easy comparison.

The original allocator functions in a very simple way: for every new variable used or every new *helper_var* created to represent an instruction in Elixir we create a new variable in the program. So the memory consumption of the original is the equivalent as the "total" memory needed for the program.

Figure 6 shows that the memory consumption of our benchmark programs vary a lot, but in all 24 cases Smart_Alloc had better or equal performance, being able to optimize bigger programs really well.

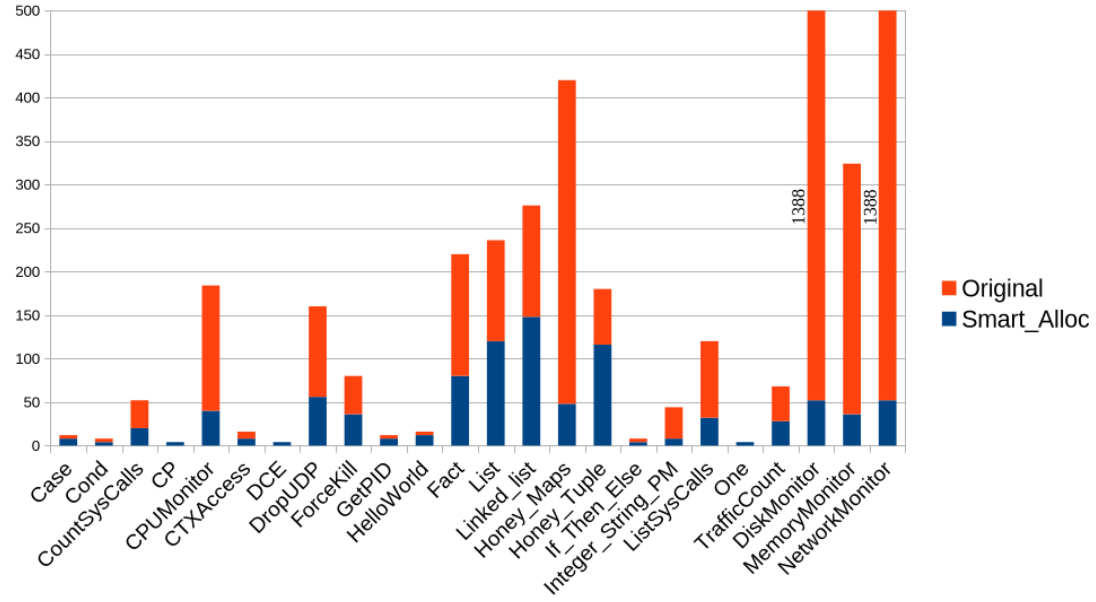

Figure 6: Absolute memory consumption in bytes.

Figure 7 showcases an easier graph to read how much the new memory allocator has reduced the new programs when compared to the original.

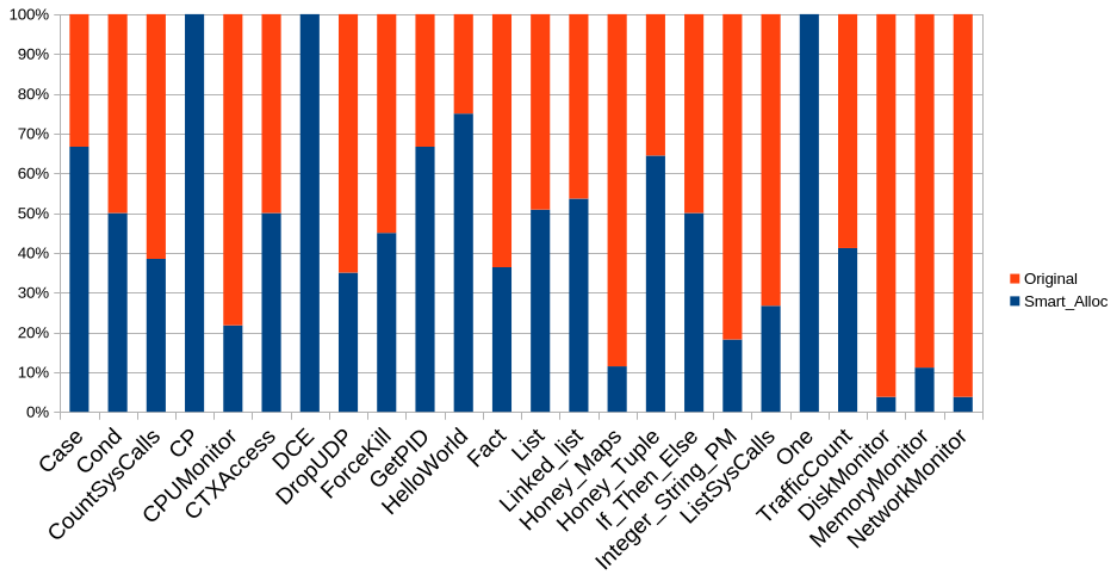

Figure 7: Percentage memory consumption in bytes.

With the exception of CP, DCE and One, which are single variable programs (4 bytes), all 21 other programs had significant improvements, from 75% up to less than 4% in Disk and Network monitors, with an average improvement of 44.6% across all programs.

4.3. Future work

Most compilers are limited by the fact that they deal with programs that have loops that may extend arbitrarily long. Because of this, it is common practice to consider only sections of code or information at a time when optimizing, for example basic blocks. This also is necessary given that more information means slower compilation.

However, Honey Potion doesn't suffer from either of those restrictions, having an Ahead-Of-Time (AOT) compilation and having programs that are finite and that can be fully unrolled for optimizations. This means that we can study new possibilities for optimizations within restricted programs.

It is possible that future languages give the user the possibility to create a restricted section where they are bounded by restrictions similar to eBPF, but that allow them heavier optimizations. Thus, researching this possibility here can also be beneficial to other languages and areas, not just the Elixir to restricted C translation.

The idea of restricting common language features for optimizations isn't new, in fact many languages differ in scope and ideas exactly to take advantage of certain metrics. Such as Rust with Memory Safety, C with raw speed, GO and other garbage collected languages with convenience and more. So it is not unexpected that some languages end up using these optimizations or that some compilers may discover sections of code that happen to respect the restrictions dealt with in this work.

In the realm of Honey Potion further studies may be able to map the memory in an ideal pattern or be able to gather important program information such as "How many defragmentations to achieve an optimal memory state".

Honey Potion has been an excellent opportunity to share, discuss and develop a compiler-based solution to the problem of eBPF accessibility. Thousands of people have seen the Honey Potion Guides and many have shown interest in the compilers laboratory or research thanks to it. This project improved on that by teaching more about memory allocation and giving good results.

References

[MD15] Stefan Marr and Stéphane Ducasse. "Tracing vs. partial evaluation: comparing meta-compilation approaches for self-optimizing interpreters". In: *SIGPLAN Not.* 50.10 (Oct. 2015), pp. 821–839. ISSN: 0362-1340. DOI: 10.1145/2858965.2814275. URL: https://doi.org/10.1145/2858965.2814275.

[Vie+20] Marcos A. M. Vieira et al. "Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications". In: *ACM Comput. Surv.* 53.1 (Feb. 2020). ISSN: 0360-0300. DOI: 10.1145/3371038. URL: https://doi.org/10.1145/3371038.

[Vis+23] Harishankar Vishwanathan et al. "Verifying the Verifier: eBPF Range Analysis Verification". In: *Computer Aided Verification*. Ed. by Constantin Enea and Akash Lal. Cham: Springer Nature Switzerland, 2023, pp. 226–251. ISBN: 978-3-031-37709-9.

[Aug+25] Kael Soares Augusto et al. "Honey Potion: An eBPF Backend for Elixir". In: *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. CGO '25. Las Vegas, NV, USA: Association for Computing Machinery, 2025, pp. 660–674. ISBN: 9798400712753. DOI: 10.1145/3696443.3708923. URL: https://doi.org/10.1145/3696443.3708923.

[Dwc] Dwctor. *Dwctor's Honey Potion Playlist*. URL: https://www.youtube.com/playlist?list=PL9cmSHf85lF5HzCha020qegkKQ3GpiEBY. Accessed 2025-11-30.

[eBP] eBPF. *eBPF Case Studies*. URL: https://ebpf.io/case-studies/. Accessed 2025-04-22.

[Lab] Compiler's Laboratory. *Honey Potion GitHub*. URL: https://github.com/lac-dcc/honey-potion. Accessed 2025-11-30.