Candelabra: Efficient selection of ideal container implementations

Aria Shrimpton

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2024

Abstract

Almost every program makes extensive use of container types – structures that hold a collection of values together. Despite many programming languages offering a variety of implementations, most programmers stick to one or two, potentially leaving large performance improvements on the table.

We present Candelabra, a system for selecting the best implementation of a container type based on the individual program's requirements. Using the DSL proposed in Qin et al. (2023), developers specify the way a container must behave and what operations it must be able to perform. Once they have done this, we are able to select implementations that meet those requirements, and suggest which will be the fastest based on the usage patterns of the user's program.

Our system is designed with flexibility in mind, meaning it is easy to add new container implementations and operations. It is also able to scale up to larger programs, without suffering the exponential blowup in time taken that would happen with a brute-force approach.

Our approach generates accurate estimates of each implementation's performance, and is able to find the fastest implementation in the majority of our tests.

We also investigate the feasibility of adaptive containers, which switch implementation once the size reaches a certain threshold. In doing so, we identify several key concerns that future work should address.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Aria Shrimpton*)

Acknowledgements

I'd like to express my deepest gratitude to my supervisor, Liam O' Connor, for his help.

I'd also like to thank the Tardis Project for the compute resources used for benchmarking, and the members of CompSoc for their advice.

Table of Contents

Chapter 1

Introduction

Almost every program makes extensive use of container data types – structures which hold a collection of values. Often, programmers will also have some requirements they want to impose on this collection, such as not storing duplicate elements, or storing the items in sorted order.

However, implementing these collection types wastes time, and can be hard to do right for more complicated structures. Most programmers will instead use one or two of the collection types provided by their language. Some languages, such as Python, go a step further, providing built-in implementations of common data structures, with special syntax and handling.

However, the underlying implementation of container types which function the same can have a drastic effect on performance (L. Liu and S. Rus (2009), Jung et al. (2011)). By largely ignoring the performance characteristics of their implementation, programmers may be missing out on large performance gains.

We propose a system, Candelabra, for the automatic selection of container implementations, based on both user-specified requirements and inferred requirements for performance. In our testing, we are able to accurately select the best performing containers for a program in significantly less time than brute force.

We have designed our system with flexibility in mind: adding new container implementations requires little effort. It is easy to adopt our system incrementally, and we integrate with existing tools to make doing so easy. The time it takes to select containers scales roughly linearly, even in complex cases, allowing our system to be used even on larger projects.

We are also able to suggest adaptive containers: containers which switch from one underlying implementation to another once they get past a certain size. Whilst we saw reasonable suggestions in our test cases, we found important performance concerns which future work could improve on.

In chapter 2, we give a more thorough description of the container selection problem, and examine previous work. We outline gaps in existing literature, and how we aim to contribute.

Chapter 3 explains the design of our solution, and how it fulfills the aims set out in chapter 2. Chapter 4 expands on this, describing the implementation work in detail and the challenges faced.

We evaluate the effectiveness of our solution in chapter 5, and identify several shortcomings that future work could improve upon.

Chapter 2

Background

In this chapter, we explain the problem of container selection and its effect on program correctness and performance. We then provide an overview of approaches taken by modern programming languages and existing literature. Finally, we explain how our system is novel, and the weaknesses in existing literature it addresses.

2.1 Container Selection

A container data type is simply a structure which holds a collection of related values. This could include a list (growable or otherwise), a set (with no duplicate elements), or something more complex (like a min heap).

In many languages, the standard library provides implementations of various container types, with users able to choose which is best for their program. This saves users a lot of time, however selecting the best type is not always straightforward.

Consider a program which needs to store and query a set of numbers, and doesn't care about ordering or duplicates. If the number of items (n) is small enough, it might be fastest to use a dynamically-sized array (known as a vector in many languages), and scan through each time we want to check if a number is inside. On the other hand, if the set we deal with is much larger, we might instead use a hash set, which provides roughly the same lookup speed regardless of size at the cost of being slower overall.

In this case, there are two factors driving our decision. Our *functional requirements* – that we don't care about ordering or duplicates – and our *non-functional requirements* – that we want our program to use resources efficiently.

2.1.1 Functional requirements

Functional requirements tell us how the container must behave in order for the program it's used in to function correctly. Continuing with our previous example, we'll compare Rust's `Vec` implementation (a dynamic array), with its `HashSet` implementation (a hash table).

To start with, we can see that the two types have different methods. `Vec` has a `.get()` method, while `HashSet` does not. If we were building a program that needed an ordered collection, replacing `Vec` with `HashSet` would likely cause the compiler to raise an error.

We will call the operations that a container implementation provides the *syntactic properties* of the implementation. In object-oriented programming, we might say they must implement an interface, while in Rust, we would say that they implement a trait.[1]

However, syntactic properties alone are not always enough to select an appropriate container implementation. Suppose our program only requires a type with the `.contains(value)` and `.len()` methods. Both `Vec` and `HashSet` satisfy these requirements, but our program might also rely on the count returned from `.len()` including duplicates. In this case, `HashSet` would give us different behaviour, causing our program to behave incorrectly.

Therefore, we also say that a container implementation has *semantic properties*. Intuitively we can think of this as the conditions upheld by the container. A `HashSet`, would have the property that there are never any duplicates. A `Vec` would not have this property, but would have the property that insertion order is preserved.

To select a correct container implementation, we then need to ensure we meet some syntactic and semantic requirements specific to our program. So long as we specify our requirements correctly, and use an implementation which provides all of the properties we're looking for, our program shouldn't be able to tell the difference.

2.1.2 Non-functional requirements

While meeting our program's functional requirements should ensure that it runs correctly, this doesn't say anything about our program's efficiency. We also want to choose the most efficient implementation available, striking a balance between runtime and memory usage.

Prior work has demonstrated that changing container implementation can give substantial performance improvements. Perflint (L. Liu and S. Rus, 2009) found and suggested fixes for "hundreds of suboptimal patterns in a set of large C++ benchmarks," with one such case improving performance by 17%. Similarly, Brainy (Jung et al., 2011) found a 27-33% speedup of real-world applications and libraries using a similar approach.

If we can find a set of implementations that satisfy our functional requirements, then one obvious solution is to benchmark the program with each of these implementations in place. This will obviously work, so long as our benchmarks are roughly representative of the real world.

Unfortunately, this technique scales poorly for larger applications. As the number of types we must select increases, the number of combinations we have to try increases exponentially.

[1] Rust's traits are most similar to typeclasses from Haskell and other FP languages. Interfaces and typeclasses have important differences, but they are irrelevant in this case.

2.2 Prior art

In this section we outline existing methods for container selection, in both current programming languages and literature.

2.2.1 Approaches in common programming languages

Modern programming languages broadly take one of two approaches to container selection.

Some languages, usually higher-level ones, recommend built-in structures as the default, using implementations that perform well enough for the vast majority of use-cases. A popular example is Python, which uses dynamic arrays as its built-in list implementation.

This approach prioritises developer ergonomics: programmers do not need to think about how these are implemented. Often other implementations are possible, but are used only when needed and come at the cost of code readability.

In other languages, collections are given as part of a standard library or must be written by the user. Java comes with growable lists as part of its standard library, as does Rust. In both cases, the standard library implementation is not special – users can implement their own and use them in the same ways.

Interfaces, or their closest equivalent, are often used to abstract over similar collections. In Java, ordered collections implement the interface `List<E>`, with similar interfaces for `Set<E>`, `Queue<E>`, etc. This allows most code to be implementation-agnostic, with functional requirements specified by the interface used.

While this provides some flexibility, it still requires the developer to choose a concrete implementation at some point. In most cases, developers will simply choose the most common implementation and assume it will be fast enough.

Otherwise, developers are forced to guess based on their knowledge of specific implementations and their program's behaviour. For more complex programs or data structures, it can be difficult or impossible to reason about an implementation's performance.

2.2.2 Rules-based approaches

One way to address this is to allow the developer to make the choice initially, but attempt to detect cases where the wrong choice was made. Chameleon (Shacham et al., 2009) is one system which uses this approach.

First, it collects statistics from program benchmarks using a "semantic profiler". This includes the space used by collections over time and the counts of each operation performed. These statistics are tracked per individual collection allocated and then aggregated by 'allocation context' — the call stack at the point where the allocation occured.

These aggregated statistics are passed to a rules engine, which uses a set of rules to

identify cases where a different container implementations might perform better. This results in a flexible engine for providing suggestions which can be extended with new rules and types as necessary. A similar approach is used by L. Liu and S. Rus (2009) for the C++ standard library.

By using the developer's selection as a baseline, both of these tools function similarly to a linter, which the developer can use to catch mistakes and suggest improvements. This makes it easy to integrate into existing projects and workflows.

However, the use of suggestion rules means that adding a new container implementations requires writing new suggestion rules. This requires the developer to understand all of the existing implementations' performance characteristics, and how they relate to the new implementation. In effect, the difficulty of selecting an implementation is offloaded to whoever writes the suggestion rules.

To ensure that functional requirements are satisfied, both systems will only suggest implementations that behave identically to the existing one. This results in selection rules being more restricted than necessary. For instance, a rule could not suggest a `HashSet` instead of a `Vec`, as the two are not semantically identical.

CoCo (Xu, 2013) and Österlund and Löwe (2013) use similar techniques, but work as the program runs. This was shown to work well for programs with different phases of execution, such as loading and then working on data. However, the overhead from profiling and from checking rules may not be worth the improvements in other programs, where access patterns are roughly the same throughout.

2.2.3 ML-based approaches

Brainy (Jung et al., 2011) gathers similar statistics, but uses machine learning for selection instead of programmed rules.

ML has the advantage of being able to detect patterns a human may not be aware of. For example, Brainy takes into account statistics from hardware counters, which are difficult for a human to reason about. This also makes it easier to add new collection implementations, as rules do not need to be written by hand.

Whilst this offers increased flexibility, it comes at the cost of requiring a more lengthy model training process when implementations are changed.

2.2.4 Estimate-based approaches

CollectionSwitch (Costa and Andrzejak, 2018) also avoids forcing developers to write rules, by estimating the performance characteristics of each implementation individually.

First, a performance model is built for each container implementation. This gives an estimate of some cost dimensions for each operation at a given collection size. The originally proposed cost dimensions were memory usage and execution time.

The system then collects data on how the program uses containers as it runs, and combines this with the built cost models to estimate the performance impact for each

collection type. It may then decide to switch between container types if the potential change in cost seems high enough. For instance, we may choose to switch if we reduce the estimated space cost by more than 20%, so long as the estimated time cost doesn't increase by more than 20%.

By generating a cost model based on benchmarks, CollectionSwitch manages to be more flexible than rule-based approaches. Like ML approaches, adding new implementations requires little extra work, but has the advantage of being possible without having to re-train a model.

2.2.5 Functional requirements

Most of the approaches we have highlighted focus on non-functional requirements, and use programming language features to enforce functional requirements. We will now examine tools which focus on container selection based on functional requirements.

Primrose (Qin et al., 2023) is one such tool, which uses a model-based approach. It allows the application developer to specify semantic requirements using a Domain-Specific Language (DSL), and syntactic requirements using Rust's traits.

Semantic requirements are expressed as a list of predicates, each representing a semantic property. Predicates act on an abstract model of the container type. Each implementation specifies how it works on this abstract model, and a constraint solver checks if the two will always agree.

This allows developers to express any combination of semantic requirements, rather than limiting them to common ones (as in Java's approach). It can also be extended with new implementations as needed, though this does require modelling the semantics of the new implementation.

Franke et al. (2022) uses an idea more similar to Java's standard library, where properties are defined by the library authors and container implementations opt in to providing them.

To select the final container implementation, both tools rely on benchmarking each candidate. As we note above, this scales poorly.

2.3 Contribution

Of the tools presented, none are designed to deal with both functional and non-functional requirements well. Our contribution is a system for container selection that addresses both of these aspects.

Users are able to specify their functional requirements in a way that is expressive enough for most usecases, and easy to integrate with existing projects. We then find which implementations in our library satisfy these requirements, and estimate which will have the best performance.

We also aim to make it easy to add new container implementations, and for our system to scale up to large projects without selection time becoming an issue.

Whilst the bulk of our system is focused on offline selection (done before the program is compiled), we also attempt to detect when changing implementation at runtime is desirable, a technique which has largely only been applied to higher-level languages.

Chapter 3

Design

We now outline the design of our container selection system (Candelabra), and justify our design decisions. We first restate our aims and priorities for the system, illustrating its usage with an example. We then provide an overview of the container selection process, and each part in it, although we leave detailed discussion of implementation for chapter 4.

3.1 Aims & Usage

As mentioned previously, we aim to create an all-in-one solution for container selection that takes into account both functional and non-functional requirements. Flexibility is a high priority: It should be easy to add new container implementations, and to integrate our system into existing applications. Our system should also be able to scale to larger programs, and remain convenient for developers to use.

We chose to implement our system as a CLI, and to work on programs written in Rust. We chose Rust both for the expressivity of its type system, and its focus on speed and low-level control. However, most of the techniques we use are not tied to Rust in particular, and so should be possible to generalise to other languages.

We require the user to provide their own benchmarks, which should be representative of a typical application run - without this, we have no consistent way to evaluate speed.

Users specify their functional requirements by listing the required traits and properties they need for a given container type. Traits are Rust's primary method of abstraction, and are similar to interfaces in object-oriented languages, or typeclasses in functional languages. Properties are specified in a Lisp-like DSL as a predicate on a model of the container.

For example, Listing 3.1 shows code from our test case based on the sieve of Eratosthenes (`src/tests/prime_sieve` in the source artifacts).

Here we request two container types: `Sieve` and `Primes`. The first must implement the `Container` and `Stack` traits, and must satisfy the `lifo` property. This property is

```
/*SPEC*
property lifo<T> {
    \c <: (Stack) -> (forall \x -> ((equal? (pop ((op-
   push c) x))) x))
}

property ascending<T> {
    \c -> ((for-all-consecutive-pairs c) leq?)
}


type Sieve<S> = {c impl (Container, Stack) | (lifo c)}
type Primes<S> = {c impl (Container) | (ascending c)}
*ENDSPEC*/
```

Listing 3.1: Container type definitions for prime_sieve

defined at the top, and requires that for any `x`, pushing `x` then popping from the container returns `x`.

The second container type, `Primes`, must implement the `Container` trait, and must satisfy the `ascending` property. This property requires that for all consecutive `x, y` pairs in the container, `x <= y`.

Once we have specified our functional requirements and provided a benchmark, we can simply run Candelabra to select a container: `candelabra-cli -p prime_sieve select`. This command outputs the information in table 3.1 and saves the best combination of container types to be used the next time the program is run. Here, the code generated uses `Vec` as the implementation for `Sieve`, and `HashSet` as the implementation for `Primes`.

Best performing	Container Type	Implementation	Estimated cost
*	Sieve	LinkedList	14179471355
	Sieve	Vec	26151238698
	Primes	HashSet	117005368
	Primes	SortedVec	112421356
*	Primes	BTreeSet	108931859

Table 3.1: Example output from selection command

3.2 Overview of process

Our tool integrates with Rust's packaging system (Cargo) to discover the information it needs about our project. It then runs a modified version of Primrose (Qin et al., 2023) to find a list of implementations satsifying our functional requirements from a pre-built library of container implementations.

Once we have this list, we build a *cost model* for each candidate type. This allows us to get an upper bound for the runtime cost of an operation at any given n. We choose to focus only on CPU time, and disregard memory usage due to the difficulty of accurately measuring memory footprint.[1]

We then run the user-provided benchmarks, using a wrapper around any of the valid candidates to track how many times each operation is performed, and the maximum size the container reaches.

We combine this information with our cost models to estimate a total cost for each candidate, which is an upper bound on the total time taken for all container operations. At this point, we also check if an adaptive container would be better, by checking if one implementation is better performing at a lower n, and another at a higher n.

Finally, we pick the implementation with the minimum cost, and generate code which allows the program to use that implementation.

Our solution requires little user intervention, integrates well with existing workflows, and the time it takes scales linearly with the number of container types in a given project.

We now go into more detail on how each step works, although we leave some specifics until Chapter 4.

3.3 Functional requirements

As described in chapter 2, any implementation we pick must satisfy the program's functional requirements. To do this, we integrate Primrose (Qin et al., 2023) as a first step.

Primrose allows users to specify both the traits they require (syntactic properties), and the semantic properties that must be satisfied.

Each container type that we want to select an implementation for is bound by a list of traits and a list of properties (lines 11 and 12 in Listing 3.1).

In brief, Primrose works by:

- Finding all implementations in the container library that implement all required traits
- Translating any specified properties to a Rosette expression

[1] As Rust is not interpreted, we would need to hook into calls to the OS' memory allocator. This is very platform-specific, although the currently work in progress allocator API (Rust allocator working group, 2016) may make this easier in future.

- For each implementation, modelling the behaviour of each operation in Rosette, and checking that the required properties always hold

We use the code provided in the Primrose paper, with minor modifications elaborated on in Chapter 4.

After this stage, we have a list of implementations for each container type we are selecting. The command `candelabra-cli candidates` will show this output, as in Table 3.2.

Type	Candidate implementation
Primes	EagerSortedVec
Primes	HashSet
Primes	BTreeSet
Sieve	LinkedList
Sieve	Vec

Table 3.2: Usable implementations by container type for `prime_sieve`

Although we use Primrose in our implementation, the rest of our system isn't dependent on it, and it would be relatively simple to use a different approach for selecting based on functional requirements.

3.4 Cost Models

Now that we have a list of correct implementations for each container type, we need a way to understand the performance characteristics of each of them. We use an approach similar to CollectionSwitch (Costa and Andrzejak, 2018), which assumes that the main factor in how long an operation takes is the current size of the collection.

Implementations have a seperate cost model for each operation, which we obtain by executing that operation repeatedly at various collection sizes.

For example, to build a cost model for `Vec::contains`, we would create several `Vecs` of varying sizes, and find the average execution time t of `contains` at each.

We then perform linear regression, using the collection size n to predict t. In the case of `Vec::contains`, we would expect the resulting polynomial to be roughly linear.

In our implementation, we fit a function of the form $x_0 + x_1 n + x_2 n^2 + x_3 \log_2 n$, using regular least-squares fitting. Before fitting, we discard all observations that are more than one standard deviation out from the mean for a given n value.

Whilst we could use a more complex technique, in practice this is good enough: Most common operations are polynomial at worst, and more complex models are at higher risk of overfitting.

This method works well for many operations and structures, although has notable limitations. In particular, implementations which defer work from one function to another will be extremely inconsistent.

For example, `LazySortedVec` (provided by Primrose) inserts new elements at the end by default, and waits to sort the list until the contents of the list are read from (such as by using `contains`). Our cost models have no way to express that the runtime of one operation varies based on the history of previous operations.

We were unable to work around this, and so we have removed these variants from our container library. One potential solution could be to perform untimed 'warmup' operations before each operation, but this is complex because it requires some understanding of what operations will cause work to be deferred.

At the end of this stage, we are able to reason about the relative cost of operations between implementations. These models are cached for as long as our container library remains the same, as they are independent of what program the user is currently working on.

3.5 Profiling applications

We now need to collect information about how the user's application uses its container types.

As mentioned above, the ordering of operations can have a large effect on container performance. Unfortunately, tracking every container operation in order quickly becomes unfeasible. Instead, we settle for tracking the count of each operation, and the maximum size the collection reaches.

Every instance or allocation of the collection produces a separate result. We then aggregate all results for a single container type into a list of partitions.

Each partition simply stores an average value for each component of our results (maximum size and a count for each operation), along with a weight indicating how many results fell into that partition.

Results are processed as in algorithm 1.

Algorithm 1 Results processing algorithm

1: Start with an empty list of partitions
2: **for** result in results **do**
3: **if** there is a partition with an average max n value within 100 of result's maximum n **then**
4: Adjust the partition's average maximum n according to the new result
5: Adjust the partitions' average count of each operation according to the counts in the new result
6: Add 1 to the weight of the partition.
7: **else**Add a new partition with the values from the result, and with weight 1.
8: **end if**
9: **end for**
10: Once all results have been processed, normalize the partition weights by dividing each by the sum of all weights.

The use of partitions serves 3 purposes. The first is to compress the data, which speeds up processing and stops us running out of memory in more complex programs. The second is to capture the fact that the number of operations will likely depend on the size of the container. The third is to aid in searching for adaptive containers, a process which relies on understanding the different sizes of containers in the application.

3.6 Selection process

At this stage, we have an estimate of how long each operation may take (from our cost models), and how often we use each operation (from our profiling information). We now combine these to estimate the total cost of each implementation. For each implementation, our estimate for its total cost is:

$$\sum_{o \in ops, (r_o, N, W) \in partitions} C_o(N) r_o W$$

- $C_o(N)$ is the cost estimated by the cost model for operation o at n value N
- r_o is the average count of a given operation in a partition
- N is the average maximum N value in a partition
- W is the weight of a partition, proportional to how many results fell in to this partition

Essentially, we scale an estimated worst-case cost of each operation by how frequently we think we will encounter it. This results in a pessimistic estimate of how much time we will spend in total on container operations for a given implementation.

Now we simply pick the implementation with the smallest estimated cost that satisfies our functional requirements for that container type.

This process is repeated for each container type in the project, and provides not only suggestions but rankings for each type. We are able to do this while only running our user's benchmarks once, which is ideal for larger applications or for long-running benchmarks.

3.7 Adaptive containers

The above process forms the core of our system, and is sufficient for normal container selection. However, a common situation in many programs is that the maximum size of a container type depends on the size of some input. In these cases, the user may write benchmarks for a range of sizes, and look for a container type that achieves good enough performance throughout the whole range.

An alternative approach is to start off with whichever container type is best at small sizes, and switch to one more suited for large amounts of data once we grow past a certain threshold. In theory, this allows the best of both worlds: A lower overhead

container when the program is being run on a small input, and a more complex one for longer runs.

For example, if a program requires a set, then for small sizes it may be best to keep a sorted list and use binary search for `contains` operations. This is what we do in our `SortedVecSet` container implementation. But when the size of the container grows, the cost of doing `contains` may grow high enough that using a `HashSet` would actually be faster.

Adaptive containers attempt to address this need, by starting off with one implementation (referred to as the low or before implementation), and switching to a new implemenation (the high or after implementation) once the size of the container passes a certain threshold.

This is similar to systems such as CoCo (Xu, 2013) and Österlund and Löwe (2013). However, we decide when to switch container implementation before the program is run, rather than as it is running. We also do so in a way that requires no knowledge of the implementation internals.

After regular container selection is done, we attempt to suggest an adaptive implementation for each container type. We first sort our list of partitions by ascending container size. If we can split our partitions in half such that everything to the left performs best with one implementation, and everything to the right with another, then we should be able to switch implementation around that n value.

In practice, finding the correct threshold is more difficult: We must take into account the cost of transforming from one implementation to another. If we adapt our container too early, we may do more work adapting it than we save if we just stuck with our low implementation. If we adapt too late, we have more data to move and less of our program gets to take advantage of the new implementation. We choose the relatively simple strategy of switching halfway between two partitions.

Our cost models allow us to estimate how expensive switching implementations will be. We compare this estimate to how much better the high implementation is than the low one, to account for the overhead of changing implementations.

A full explanation of our algorithm is given in section 4.4.

Chapter 4

Implementation

We now elaborate on our implementation, explaining some of the finer details of our design. With reference to the source code, we explain the structure of our system's implementation, and highlight areas with difficulties.

4.1 Modifications to Primrose

In order to facilitate integration with Primrose, we refactored large parts of the code to support being called as an API, rather than only through the command line. This also required updating the older code to a newer edition of Rust, and improving the error handling throughout.

As suggested in the original paper, we added the ability to deal with associative container types: key to value mappings. We added the `Mapping` trait to the implementation library, and updated the type checking and analysis code to support multiple type variables.

Operations on mapping implementations can be modelled and checked against constraints in the same way that regular containers can be. They are modelled in Rosette as a list of key-value pairs. `src/crates/library/src/hashmap.rs` shows how mapping container types can be declared, and operations on them modelled.

Table 4.1 shows the library of container types we used. Most come from the Rust standard library, with the exceptions of the `SortedVec` family of containers, which use `Vec` internally. The library source can be found in `src/crates/library`.

We also added new syntax to Primrose's domain-specific language to support defining properties that only make sense for mappings (`dictProperty`), however this was unused.

While performing integration testing, we found and fixed several other issues with the existing code:

1. Only push and pop operations could be modelled in properties. Other operations would raise an error during type-checking.
2. The Rosette code generated for properties using other operations was incorrect.

Implementation	Description
`LinkedList`	Doubly-linked list
`Vec`	Contiguous growable array
`VecSet`	Vec with no duplicates
`VecMap`	A Vec of (K, V) tuples, used as a Mapping
`SortedVec`	Vec kept in sorted order
`SortedVecSet`	Vec kept in sorted order, with no duplicates
`VecMap`	A Vec of (K, V) tuples sorted by key, used as a Mapping
`HashMap`	Hash map with quadratic probing
`HashSet`	Hash map with empty values
`BTreeMap`	B-Tree (Bayer and McCreight, 1970) map with linear search.
`BTreeSet`	B-Tree map with empty values

Table 4.1: Implementations in our library

3. Some trait methods used mutable borrows unnecessarily, making it difficult or impossible to write safe Rust using them.
4. The generated code would perform an unnecessary heap allocation for every created container, which could affect performance.

We also added requirements to the `Container` and `Mapping` traits related to Rust's `Iterator` API. Among other things, this allows us to use for loops, and to more easily move data from one implementation to another.

4.2 Building cost models

In order to benchmark container types, we use a seperate crate containing benchmarking code for each trait in the Primrose library (`src/crates/benchmarker`). When benchmarks need to be run for an implementation, we dynamically generate a new crate, which runs all benchmark methods appropriate for the given implementation (`src/crate/candelabra/src/cost/benchmark.rs`).

As Rust's generics are monomorphised, our generic code is compiled as if we were using the concrete type in our code, so we don't need to worry about affecting the benchmark results.

Each benchmark is run in a 'warmup' loop for a fixed amount of time (currently 500ms), then runs for a fixed number of iterations (currently 50). This is important because we are using least squares fitting - if there are less data points at higher n values then our resulting model may not fit those points as well. We repeat each benchmark at a range of n values: $10, 50, 100, 250, 500, 1000, 6000, 12000, 24000, 36000, 48000, 60000$.

Each benchmark we run corresponds to one container operation. For most operations, we insert n random values to a new container, then run the operation once per iteration. For certain operations which are commonly amortized (`insert`, `push`, and `pop`), we instead run the operation itself n times and divide all data points by n.

As discussed previously, we discard all points that are outwith one standard deviation of the mean for each n value. We use the least squares method to fit a polynomial of form $x_0 + x_1 n + x_2 n^2 + x_3 \log_2 n$.

As most operations on common data structures are polynomial or logarithmic complexity, we believe that this function is good enough to capture the cost of most operations. We originally experimented with coefficients up to x^3, but found that this led to overfitting.

4.3 Profiling

We implement profiling using the `ProfilerWrapper` type (`src/crates/library/src/profiler.rs`), which takes as type parameters the inner container implementation and an index, used later to identify what container type the output corresponds to. We then implement any Primrose traits that the inner container implements, counting the number of times each operation is called. We also check the length of the container after each insert operation, and track the maximum.

Tracking is done per-instance, and recorded when the container goes out of scope and its `Drop` implementation is called. We write the counts of each operation and maximum size of the collection to a location specified by an environment variable.

When we want to profile a program, we pick any valid inner implementation for each selection site, and use that candidate with our profiling wrapper as the concrete implementation for that site. We then run all of the program's benchmarks once, which gives us an equal sample of data from each of them.

This approach has the advantage of giving us information on each individual collection allocated, rather than only statistics for the type as a whole. For example, if one instance of a container type is used in a very different way from the rest, we will be able to see it more clearly than a normal profiling tool would allow us to.

Although there is noticeable overhead in our current implementation, this is not important as we aren't measuring the program's execution time when profiling. We could likely reduce profiling overhead by batching file outputs, however this wasn't necessary for us.

4.4 Container Selection

Selection is done per container type. For each candidate implementation, we calculate its cost on each partition in the profiler output, then sum these values to get the total estimated cost for each implementation. This is implemented in `src/crates/candelabra/src/profiler/info.rs` and `src/crates/candelabra/src/select.rs`.

In order to try and suggest an adaptive container, we use algorithm 2.

Algorithm 2 Adaptive container suggestion algorithm

Sort *partitions* in order of ascending maximum n values.
costs ← the cost for each candidate in each partition.
best ← the best candidate per partition
$i \leftarrow$ the lowest index where $best[i] \neq best[0]$
if exists $j < i$ where $best[j] \neq best[0]$, or exists $j \geq i$ where $best[j] \neq best[i]$ **then**
    return no suggestion
end if
before ← name of the candidate in `best[0]`
after ← name of the candidate in `best[i]`
threshold ← halfway between the max n values of partition i and $i-1$
$switching\_cost \leftarrow C_{before,clear}(threshold) + threshold * C_{after,insert}(threshold)$
not_switching_cost ← the sum of the difference in cost between *before* and *after* for all partitions with index $> i$.
if $switching\_cost > not\_switching\_cost$ **then**
    return no suggestion
else
    return suggestion for an adaptive container which switches from *before* to *after* when n gets above *threshold*. Its estimated cost is $switching\_cost + \sum_{k=0}^{i} costs[before][k] + \sum_{k=i}^{|partitions|} costs[after][k]$
end if

4.5 Code Generation

As mentioned in chapter 3, we made modifications to Primrose's code generation in order to improve the resulting code's performance. The original Primrose code would generate code as in Listing 4.1. In order to ensure that users specify all of the traits they need, this code only exposes methods on the implementation that are part of the trait bounds given. However, it does this by using a `dyn` object, Rust's mechanism for dynamic dispatch.

Although this approach works, it adds an extra layer of indirection to every call: The caller must use the dyn object's vtable to find the method it needs to call. This also prevents the compiler from optimising across this boundary.

In order to avoid this, we make use of Rust's support for existential types: Types that aren't directly named, but are inferred by the compiler. Existential types only guarantee their users the given trait bounds, therefore they accomplish the same goal of forcing users to specify all of their trait bounds upfront.

Figure 4.2 shows our equivalent generated code. The type alias `Stack<S>` only allows users to use the `Container<S>`, `Stack<S>`, and `Default` traits. Our unused 'dummy' function `_StackCon` has the return type `Stack<S>`. Rust's type inference step sees that its actual return type is `Vec<S>`, and therefore sets the concrete type of `Stack<S>` to `Vec<S>` at compile time.

Unfortunately, this feature is not yet in stable Rust, meaning we have to opt in to it using

```
pub trait StackTrait<T> : Container<T> + Stack<T> {}
impl<T: 'static + Ord + std::hash::Hash> StackTrait<T>
    for <Stack<T> as ContainerConstructor>::Impl {}


pub struct Stack<T> {
    elem_t: core::marker::PhantomData<T>,
}

impl<T: 'static + Ord + std::hash::Hash>
    ContainerConstructor for Stack<T> {
    type Impl = Vec<T>;
    type Bound = dyn StackTrait<T>;
    fn new() -> Box<Self::Bound> {
        Box::new(Self::Impl::new())
    }
}
```

Listing 4.1: Code generated by original Primrose project

```
pub type StackCon<S: PartialEq + Ord + std::hash::Hash> =
    impl Container<S> + Stack<S> + Default;

#[allow(non_snake_case)]
fn _StackCon<S: PartialEq + Ord + std::hash::Hash>() ->
    StackCon<S> {
    std::vec::Vec::<S>::default()
}
```

Listing 4.2: Code generated with new method

an unstable compiler flag (`feature(type_alias_impl_trait)`). At time of writing, the main obstacle to stabilisation appears to be design decisions that only apply to more complicated use-cases, therefore we are confident that this code will remain valid and won't encounter any compiler bugs.

Chapter 5

Results & analysis

In this chapter, we present the methodology used for benchmarking our system, our results, and analysis. We examine the produced cost models of certain operations in detail, with reference to the expected asymptotics of each operation. We then compare the selections made by our system to the actual optimal selections (obtained by brute force) for a variety of test cases. This includes examining when adaptive containers are suggested, and their effectiveness.

5.1 Testing setup

In order to ensure consistent results and reduce the effect of other running processes, all benchmarks were run on a KVM virtual machine on server hardware. We used 4 cores of an Intel Xeon E5-2687Wv4 CPU, and 4GiB of RAM.

The VM was managed and provisioned using NixOS, meaning it can be easily reproduced with the exact software we used. Instructions on how to do so are in the supplementary materials. The most important software versions are listed below.

- Linux 6.1.64
- Rust nightly 2024-01-25
- LLVM 17.0.6
- Racket 8.10

5.2 Cost models

We start by examining some of our generated cost models, comparing them both to the observations they are based on, and what we expect from asymptotic analysis. As we build a total of 77 cost models from our library, we will not examine them all in detail. We look at models of the most common operations, grouped by containers that are commonly selected together.

5.2.1 Insert operations

Starting with the `insert` operation, Figure 5.1 shows how the estimated cost changes with the size of the container. The lines correspond to our fitted curves, while the points indicate the raw observations we drew from.

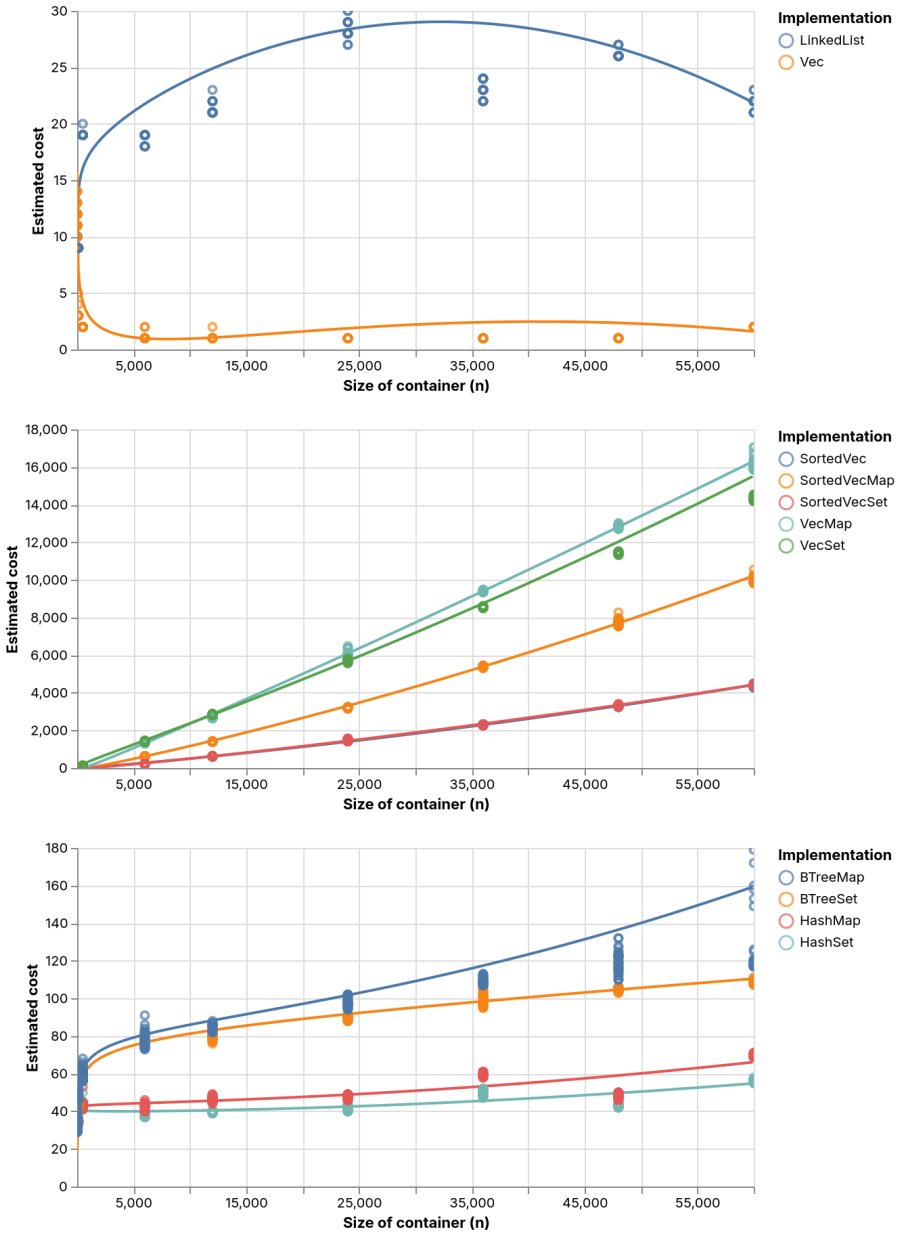

Figure 5.1: Estimated cost of insert operation by implementation

Starting with `Vec`, we see that insertion is very cheap, and gets slightly cheaper as the size of the container increases. This roughly agrees with the expected $O(1)$ time of amortised inserts on a Vec. However, we also note a sharply increasing curve when n is small, and a slight 'bump' around $n = 35,000$. The former appears to be in line with the observations, and is likely due to the static growth rate of Rust's Vec implementation. The latter appears to diverge from the observations, and may indicate poor fitting.

`LinkedList` has a significantly slower insertion. This is likely because it requires a heap allocation system call for every item inserted, no matter the current size. This would also explain why data points appear spread out more, as system calls have more unpredictable latency, even on systems with few other processes running. Notably, insertion appears to start to get cheaper past $n = 24,000$, although this is only weakly suggested by observations.

It's unsurprising that these two implementations are the cheapest, as they have no ordering or uniqueness guarantees, unlike our other implementations.

The `SortedVec` family of containers (`SortedVec`, `SortedVecSet`, and `SortedVecMap`) all exhibit roughly logarithmic growth, with `SortedVecMap` exhibiting a slightly higher growth rate. This is expected, as internally all of these containers perform a binary search to determine where the new element should go, which is $O(\lg n)$ time.

`SortedVecMap` exhibits roughly the same shape as its siblings, but with a slightly higher growth rate. This pattern is shared across all of the `*Map` types we examine, and could be explained by the increased size of each element reducing the effectiveness of the cache.

`VecMap` and `VecSet` both have a significantly higher, roughly linear, growth rate. Both of these implementations work by scanning through the existing array before each insertion to check for existing keys, therefore a linear growth rate is expected.

`HashSet` and `HashMap` insertions are much less expensive, and mostly linear with only a slight growth at very large n values. This is what we expect for hash-based collections, with the slight growth likely due to more hash collisions as the size of the collection increases.

`BTreeSet` has similar behaviour, but settles at a larger value overall. `BTreeMap` appears to grow more rapidly, and cost more overall.

It's important to note that Rust's `BTreeSet` is not based on binary tree search, but instead a more general tree search originally proposed by Bayer and McCreight (1970), where each node contains $B-1$ to $2B-1$ elements in an unsorted array. The standard library documentation (Rust Documentation Team, 2024) states that search is expected to take $O(B \lg n)$ comparisons. Since both of these implementations require searching the collection before inserting, the close-to-logarithmic growth seems to makes sense.

5.2.1.1 Small n values

Whilst our main figures for insertion operations indicate a clear winner within each category, looking at small n values reveals more complexity. Figure 5.2 shows the cost models for insert operations on different set implementations at smaller n values.

Note that for $n < 1800$ the overhead from sorting a vec is less than running the default hasher function (at least on this hardware).

We also see a sharp spike in the cost for `SortedVecSet` at low n values, and an area of supposed 0 cost from around $n = 200$ to $n = 800$. This seems inaccurate, and indicates

that our current fitting procedure may not be able to deal with low n values properly. More work is required to improve this.

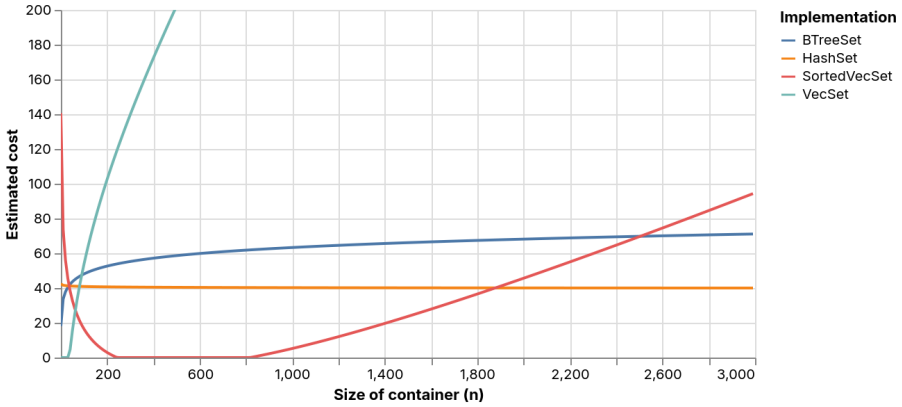

Figure 5.2: Estimated cost of insert operation on set implementations, at small n values

5.2.2 Contains operations

We now examine the cost of the `contains` operation. Figure 5.3 shows our built cost models, again grouped for readability.

The observations in these graphs have a much wider spread than our `insert` operations do. This is probably because we attempt to get a different random element in our container every time, so our observations show the best and worst case of our data structures. This is desirable assuming that `contains` operations are actually randomly distributed in the real world, which seems likely.

For the `SortedVec` family, we would expect to see roughly logarithmic growth, as we are performing a binary search. This is the case for `SortedVecMap`, however `SortedVec` and `SortedVecSet` both show exponential growth with a 'dip' around $n = 25{,}000$. It's unclear why this is, one reason could be that the elements we query are randomly distributed throughout the list, and this distribution may not be fair for all benchmarks. A possible improvement would be to run contains with a known distribution of values, including low, high, and not present values in equal parts.

The `Vec` family exhibits roughly linear growth, which is expected, since this implementation scans through the whole array each time.

`LinkedList` has roughly logarithmic growth, at a significantly higher cost. The higher cost is expected, although its unclear why growth is logarithmic rather than linear. As the spread of points also appears to increase at larger n values, its possible that this is due to larger n values causing a higher proportion of the program's memory to be dedicated to the container, resulting in better cache utilisation.

`HashSet` appears roughly linear as expected, with only a slow logarithmic rise, probably due to an increasing amount of collisions. `BTreeSet` is consistently above it, with a slightly faster logarithmic rise.

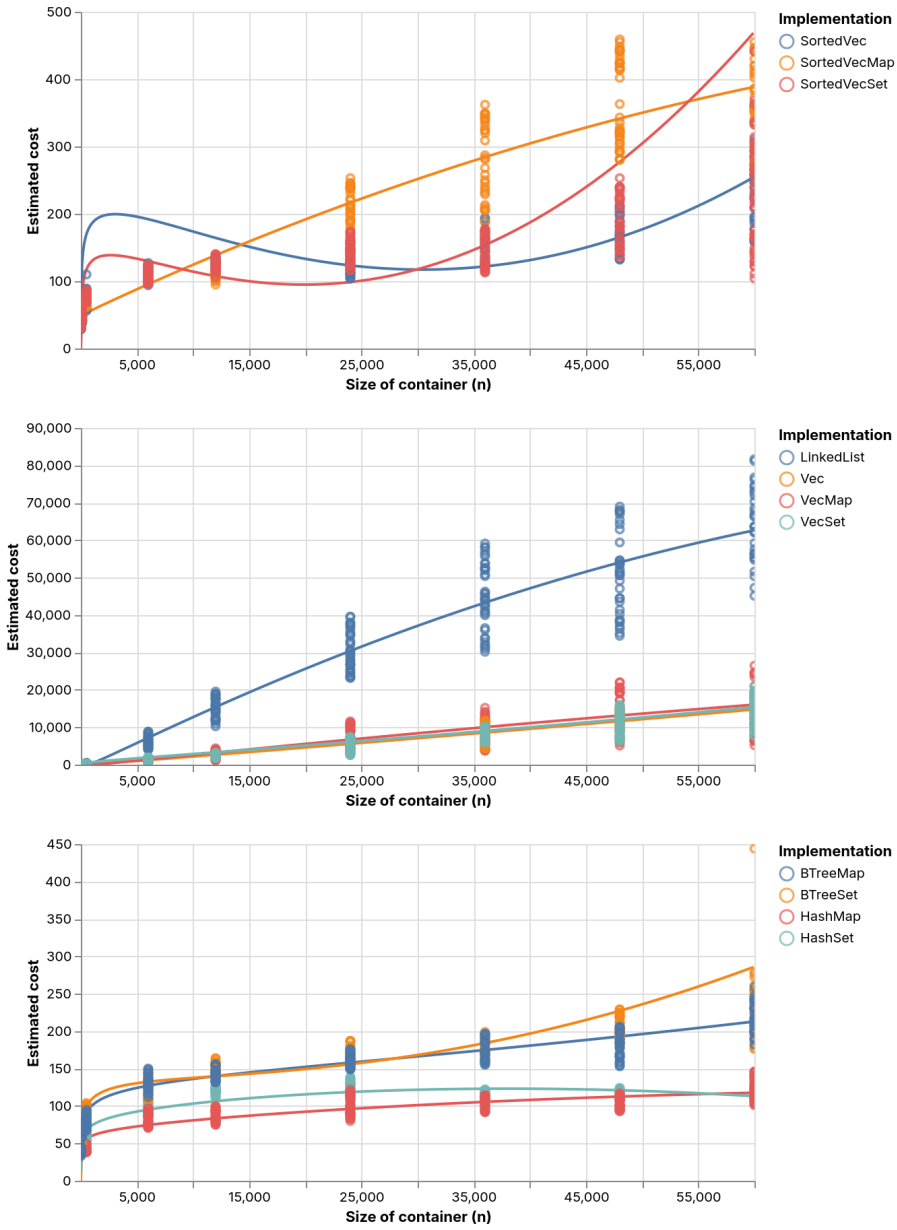

Figure 5.3: Estimated cost of `contains` operation by implementation

`BTreeMap` and `HashMap` both mimic their set counterparts, but with a slightly lower cost and growth rate. It's unclear why this is, however it could be related to the larger spread in observations for both implementations.

5.2.3 Evaluation

Overall, our cost models appear to be a good representation of each implementations performance impact. Future improvements should focus on improving accuracy at lower n values, such as by employing a more complex fitting procedure, or on ensuring operations have their best and worst cases tested fairly.

5.3 Selections

We now proceed with end-to-end testing of the system, selecting containers for a sample of test programs with varying needs.

5.3.1 Benchmarks

Our test programs broadly fall into two categories: examples programs, which repeat a few operations many times, and real-life programs, which are implementations of common algorithms and solutions to programming puzles. We expect the results from our example programs to be relatively obvious, while our real programs are more complex and harder to predict.

Most of our real programs are solutions to puzzles from Advent of Code (Wastl, 2015), a popular collection of programming puzzles. Table 5.1 lists and briefly describes our test programs.

Name	Description
example_sets	Repeated insert and contains operations on a set.
example_stack	Repeated push and pop operations on a stack.
example_mapping	Repeated insert and get operations on a mapping.
prime_sieve	Sieve of eratosthenes algorithm.
aoc_2021_09	Flood-fill like algorithm (Advent of Code 2021, Day 9)
aoc_2022_08	Simple 2D raycasting (AoC 2022, Day 8)
aoc_2022_09	Simple 2D soft-body simulation (AoC 2022, Day 9)
aoc_2022_14	Simple 2D particle simulation (AoC 2022, Day 14)

Table 5.1: Our test programs

Table 5.2 shows the difference in benchmark results between the slowest possible assignment of containers, and the fastest. Even in our example programs, we see that the wrong choice of container can slow down our programs substantially. In all but two programs, the wrong implementation can more than double the runtime.

Project	Maximum slowdown (ms)	Maximum relative slowdown
aoc_2021_09	55206.94	12.0
aoc_2022_08	12161.38	392.5
aoc_2022_09	18.96	0.3
aoc_2022_14	83.82	0.3
example_mapping	85.88	108.4
example_sets	1.33	1.6
example_stack	0.36	19.2
prime_sieve	26093.26	34.1

Table 5.2: Spread in total benchmark results by program

5.3.2 Prediction accuracy

We now compare the implementations suggested by our system to the selection that is actually best, which we obtain by brute-forcing all possible implementations. We leave analysis of adaptive container suggestions to section 5.3.3

Table 5.3 shows the predicted best assignments alongside the actual best assignment, obtained by brute-force. In all but two of our test cases (marked with *), we correctly identify the best container.

Incorrect	Project	Container Type	Best implementation	Predicted best
	aoc_2021_09	Map	HashMap	HashMap
	aoc_2021_09	Set	HashSet	HashSet
	aoc_2022_08	Map	HashMap	HashMap
	aoc_2022_09	Set	HashSet	HashSet
	aoc_2022_14	Set	HashSet	HashSet
*	aoc_2022_14	List	Vec	LinkedList
	example_mapping	Map	HashMap	HashMap
	example_sets	Set	HashSet	HashSet
	example_stack	StackCon	Vec	Vec
	prime_sieve	Primes	BTreeSet	BTreeSet
*	prime_sieve	Sieve	Vec	LinkedList

Table 5.3: Actual best vs predicted best implementations

Both of these failures appear to be caused by our system being overly eager to suggest a `LinkedList`. From looking at detailed profiling information, it seems that both of these container types had a relatively small amount of items in them. Therefore this is likely caused by our cost models being inaccurate at small n values, as mentioned in section 5.2.1.1.

Overall, our results suggest that our system is effective, at least for large enough n values. Unfortunately, these tests are somewhat limited, as the best container seems easy to predict for most cases: `Vec` where uniqueness is not important, and `Hash*` otherwise. Therefore, more thorough testing is needed to fully establish the system's effectiveness.

5.3.3 Adaptive containers

We now look at cases where an adaptive container was suggested, and evaluate the result.

Table 5.4 shows the container types for which adaptive containers were suggested, along with the inner types and the threshold at which to switch.

The suggested containers for both `aoc_2022_08` and `example_mapping` are unsurprising. Since hashing incurs a roughly constant cost, it makes sense that below a certain n value, simply searching through a list is more effective. The suggestion of `SortedVecMap` vs `VecMap` likely has to do with the relative frequency of `insert` operations compared to others.

Project	Container Type	Suggestion
aoc_2022_08	Map	SortedVecMap until n=1664, then HashMap
aoc_2022_09	Set	HashSet until n=185540, then BTreeSet
example_mapping	Map	VecMap until n=225, then HashMap
prime_sieve	Primes	BTreeSet until n=34, then HashSet
prime_sieve	Sieve	LinkedList until n=747, then Vec

Table 5.4: Suggestions for adaptive containers

The suggestion to start with a `LinkedList` for `prime_sieve / Sieve` is likely due to the same issues that cause a `LinkedList` to be suggested in the non-adaptive case. This may also be the case for the suggestion of `BTreeSet` for `prime_sieve / Primes`.

The suggestion of `BTreeSet` for `aoc_2022_09` is most surprising. As the *n* threshold after which we switch is outside the range we benchmark our implementations at, this suggestion is based on our model attempting to generalise far outside the range it has seen before.

Table 5.5 compares our adaptive container suggestions with the fastest non-adaptive implementation. Since we must select an implementation for all containers before selecting a project, we show all possible combinations of adaptive and non-adaptive container selections where appropriate.

Note that the numbered columns indicate the benchmark 'size', not the actual size that the container reaches within that benchmark. What this means exactly varies by benchmark.

In all but one project, the non-adaptive containers are as fast or faster than the adaptive containers at all sizes of benchmarks. In the `aoc_2022_09` project, the adaptive container is marginally faster until the benchmark size reaches 2000, at which point it is significantly slower.

This shows that adaptive containers as we have implemented them are not effective in practice. Even in cases where we never reach the size threshold, the presence of adaptive containers has an overhead which slows down the program 3x in the worst case (`example_mapping`, size = 150).

One explanation for this could be that every operation now requires checking which inner implementation we are using, resulting in an additional check for each operation. More work could be done to minimise this overhead, although it's unclear how.

It is also unclear if the threshold values that we suggest are the optimal ones. Currently, we decide our threshold by picking a value between two partitions with different best containers. Future work could take a more complex approach that finds the best threshold value based on our cost models, and takes the overhead of all operations into account.

5.3.4 Evaluation

Overall, we find that the main part of our container selection system has merit. While our testing has limitations, it shows that we can correctly identify the best container even in complex programs. More work is needed on improving our system's performance for very small containers, and on testing with a wider range of programs.

Our proposed technique for identifying adaptive containers appears ineffective. The primary challenges appear to be in the overhead introduced to each operation, and in finding the correct point at which to switch implementations. This could also suggest that adaptive containers are less effective in lower-level compiled languages, as previous literate focused mostly on higher-level languages such as Java (Xu, 2013; Costa and Andrzejak, 2018).

Project	Implementations	Benchmark size					
		100	200				
aoc_2022_08	Map=HashMap	1ms ± 12us	6ms ± 170us				
aoc_2022_08	Map=Adaptive	1ms ± 74us	6ms ± 138us				
		100	1000	2000			
aoc_2022_09	Set=HashSet	1ms ± 6us	10ms ± 51us	22ms ± 214us			
aoc_2022_09	Set=Adaptive	1ms ± 3us	10ms ± 27us	40ms ± 514us			
		50	150	2500	7500		
example_mapping	Map=HashMap	3us ± 6ns	11us ± 20ns	184us ± 835ns	593us ± 793ns		
example_mapping	Map=Adaptive	4us ± 9ns	33us ± 55ns	192us ± 311ns	654us ± 19us		
		50	500	50000			
prime_sieve	Primes=BTreeSet, Sieve=Vec	1us ± 2ns	75us ± 490ns	774ms ± 4ms			
prime_sieve	Primes=BTreeSet, Sieve=Adaptive	2us ± 7ns	194us ± 377ns	765ms ± 4ms			
prime_sieve	Primes=Adaptive, Sieve=Vec	1us ± 10ns	85us ± 179ns	788ms ± 2ms			
prime_sieve	Primes=Adaptive Sieve=Adaptive	2us ± 5ns	203us ± 638ns	758ms ± 4ms			

Table 5.5: Adaptive containers vs the best single container, by size of benchmark

Chapter 6

Conclusion

We have presented an integrated system for container implementation selection, which can take into account the functional and non-functional requirements of the program it is working on.

Our system is extremely flexible, and can be easily extended with new container types and new functionality on those types, as we showed by adding associative collections and several new data types to our library.

We demonstrated how benchmarking of container implementations and profiling of target applications can be done separately and then combined. We prove that this approach has merit, although our testing had notable limitations that future work should improve on. We also found that while linear regression is powerful enough for many cases, more research is required on how best to gather and preprocess data in order to best capture an implementation's performance characteristics.

We test the effectiveness of adaptive containers, in which the underlying implementation changes as the container grows. We find significant challenges in implementing this technique, suggesting that the overhead incurred is more important in lower-level compiled languages such as Rust. Future work should focus on minimising this overhead, as well as on finding the correct threshold at which to switch implementation.

Bibliography

Xueying Qin, Liam O'Connor, and Michel Steuwer. Primrose: Selecting container data types by their properties. 7(3), 2023. doi: 10.22152/programming-journal.org/2023/7/11. URL http://arxiv.org/abs/2205.09655.

L. Liu and S. Rus. Perflint: A context sensitive performance advisor for c++ programs. In *2009 International Symposium on Code Generation and Optimization*, pages 265–274, 2009. doi: 10.1109/CGO.2009.36.

Changhee Jung, Silvius Rus, Brian P. Railing, Nathan Clark, and Santosh Pande. Brainy: Effective selection of data structures. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 86–97. Association for Computing Machinery, 2011. doi: 10.1145/1993498.1993509. URL https://doi.org/10.1145/1993498.1993509.

Ohad Shacham, Martin Vechev, and Eran Yahav. Chameleon: adaptive selection of collections. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 408–418. ACM, 2009. doi: 10.1145/1542476.1542522. URL https://dl.acm.org/doi/10.1145/1542476.1542522.

Guoqing Xu. CoCo: Sound and adaptive replacement of java collections. In *ECOOP 2013 – Object-Oriented Programming*, volume 7920, pages 1–26. Springer Berlin Heidelberg, 2013. doi: 10.1007/978-3-642-39038-8_1. URL http://link.springer.com/10.1007/978-3-642-39038-8_1.

Erik Österlund and Welf Löwe. Dynamically transforming data structures. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 410–420, 2013. doi: 10.1109/ASE.2013.6693099.

Diego Costa and Artur Andrzejak. CollectionSwitch: a framework for efficient and dynamic collection selection. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 16–26. ACM, 2018. doi: 10.1145/3168825. URL https://dl.acm.org/doi/10.1145/3168825.

Björn Franke, Zhibo Li, Magnus Morton, and Michel Steuwer. Collection skeletons: Declarative abstractions for data collections. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2022, pages 189–201. Association for Computing Machinery, 2022. doi: 10.1145/3567512.3567528. URL https://doi.org/10.1145/3567512.3567528.

Rust allocator working group. Rfc 1398: Allocators, 2016. URL https://github.com/rust-lang/rfcs/blob/master/text/1398-kinds-of-allocators.md.

R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control - SIGFIDET '70*, page 107. ACM Press, 1970. doi: 10.1145/1734663.1734671. URL http://portal.acm.org/citation.cfm?doid=1734663.1734671.

Rust Documentation Team. BTreeMap documentation, 2024. URL https://doc.rust-lang.org/stable/std/collections/struct.BTreeMap.html.

Eric Wastl. Advent of code, 2015. URL https://adventofcode.com/2022/about.