Automatic Data Enumeration for Fast Collections

Tommy McMichen
Northwestern University
Evanston, Illinois, USA

Simone Campanoni*
Northwestern University
Evanston, Illinois, USA

***Abstract*—Data collections provide a powerful abstraction to organize data, simplifying development and maintenance. Choosing an implementation for each collection is a critical decision, with performance, memory and energy tradeoffs that need to be balanced for each use case. Specialized implementations offer significant benefits over their general-purpose counterparts, but also require certain properties of the data they store, such as uniqueness or ordering. To employ them, developers must either possess domain knowledge or transform their data to exhibit the desired property, which is a tedious, manual process. One such transformation—commonly used in data mining and program analysis—is data enumeration, where data items are assigned unique identifiers to enable fast equality checks and compact memory layout. In this paper, we present an automated approach to data enumeration, eliminating the need for manual developer effort. Our implementation in the MEMOIR compiler achieves speedups of 2.16× on average (up to 8.72×) and reduces peak memory consumption by 5.6% on average (up to 50.7%). This work shows that automated techniques can manufacture data properties to unlock specialized collection implementations, pushing the envelope of collection-oriented optimization.**

***Keywords*—compilers, memory optimization**

I. Introduction

When developers use data collections (e.g., lists, sets, maps), they must choose how to implement them. While custom implementations allow hand-tuned performance and memory use, many rely on general-purpose implementations provided by the language or libraries. For example, a C++ developer can select either a `std::vector` (array) or a `std::list` (linked list) to optimize common operations. This process, called *collection selection*, significantly impacts both performance and memory usage [1, 2]. Consequently, tools exist to guide developers through this optimization space [2–11].

Furthermore, collection implementations can be specialized to leverage properties of the data they store. For example, if a certain key is never used in a hashtable, it can be used as a tombstone key to mark slots that are empty or removed, eliding the need for an extra bit to mark each slot. Without such properties, selections like this are left entirely unexplored. Implicit knowledge of properties is a hallmark of domain-specific languages, naturally enabling specialized implementations. However, for general-purpose applications, these properties do not exist naturally. In these cases, developers must *manufacture* the necessary property by normalizing their data if they wish to employ a specialized implementation.

For example, bitsets provide a compact and performant solution to store finite sets over a contiguous range of integers, e.g., $[0, N)$. This property can be manufactured with *data enumeration*, where each item in the domain of a set is assigned a unique integer identifier. These identifiers can then be used as the domain of a bitset. Consider the following code, where we find and print all unique items in an array:

```
1 array := ["foo", "bar", "foo"]
2 set: Set = {}
3 for v in array:
4   if not set.has(v):
5     set.insert(v)
6     print(v)
```

This program can be transformed to use data enumeration (shown below). First, a mapping from identifiers to the original data is created (Line 1). Items in the array are replaced with their identifiers (Line 2). Then, the set is replaced with a bitset over the identifiers (Line 3). The original data is retrieved by indexing into the mapping with the identifier (Line 7).

```
1 enumn := {0->"foo", 1->"bar"}
2 array := [0, 1, 0]
3 set: BitSet = {}
4 for v in array:
5   if not set.has(v):
6     set.insert(v)
7     print(enumn[v])
```

While data enumeration may seem straightforward, it can be unwieldy to perform manually. Developers must balance the cost of translating between data and identifiers, ensuring that operations are performed on the correct form. In large projects, keeping usage consistent across modules is difficult, risking subtle logic errors, while evolving features and refactoring can introduce performance bugs as boundary choices become stale.

To ameliorate these issues, we propose *automatic data enumeration (ADE)*. With ADE, the compiler creates the enumeration mapping, inserts translations between data and identifiers, and replaces sets with bitsets. We introduce optimization techniques to eliminate redundant translations and employ static heuristics to decide when to use either identifiers or the original data. With these techniques, ADE achieves speedups of 2.16× on average (up to 8.72×) and reduces peak memory consumption by 5.6% on average (up to 50.7%) on a wide range of benchmarks. Furthermore, user directives enable performance engineering for up to 78.1× speedup and a 70.1% peak memory reduction. We implement ADE in the MEMOIR compiler [12]. The MEMOIR compiler with ADE is publicly available at https://github.com/arcana-lab/memoir.

*Now at Google.

This paper makes the following contributions:

- Introduces Automatic Data Enumeration (§III).
- Presents optimizations to eliminate redundant translations between data and identifiers (§III-C–III-E).
- Extends MEMOIR to support collection selection (§III-H).
- Enables performance engineering with directives (§III-I).
- Evaluates the performance and memory usage impact of ADE on multiple architectures (§IV).
- Investigates the impact of each optimization (§IV:RQ3).
- Explores a performance engineering case study using directives to fine tune ADE (§IV:RQ4).

II. Background and Motivations

Data enumeration is a manual optimization rooted in string interning, where each string is assigned a unique identifier. Interning is useful when strings serve as map keys or set items, avoiding an $O(n)$ multiplicative factor on all search operations. This pattern is commonly seen in compilers and language runtimes, with the earliest documented usage in Common Lisp [13] and Scheme [14]. It persists today in modern languages such as Java [15], Lua [16], and Clojure [17].

However, interning applies to more than strings. Java interns boxed booleans, and some integers, to reduce memory usage [18]. Additionally, Go's `unique` package [19] lets developers intern comparable data types, exposing a `Handle` type for identifiers. More generally, the flyweight design pattern [20] is used by developers to deduplicate shared immutable data, an example being LLVM's storage of constant values.

Data enumeration differs from interning by requiring that all unique identifiers be contiguous, i.e., in the range $[0, N)$ where N is the number of unique data items. With this property, maps and sets over the data can be converted into array-like data structures, such as bitsets, where a key's identifier is used as its index into the array. Converting sets and maps into array-like data structures is the primary benefit of data enumeration.

We build ADE on top of MEMOIR [12], a compiler intermediate representation (IR) with data collections as first class citizens in an SSA form. MEMOIR enables the collection-oriented analysis and transformation necessary for ADE. We describe our extensions to MEMOIR, along with a brief overview of technical details and syntax, in §III-A.

III. Automatic Data Enumeration

Before going into the details of automatic data enumeration (ADE), we discuss our high-level approach.

Associative collections map keys of an arbitrary type to values. While their definition affords greater flexibility than sequential collections, they require indirection—e.g., search trees or probes—to access. Fundamentally, this arises from mapping a sparse or noncontiguous domain into memory:

$$\mathbb{K} \xrightarrow{\text{sparse}} \mathbb{V}$$

We address this limitation by decomposing associative collections into a sparse mapping and a dense mapping:

$$\mathbb{K} \xrightarrow{\text{sparse}} \mathbb{E} \xrightarrow{\text{dense}} \mathbb{V}, \text{ where } \mathbb{E} = [0, |\mathbb{K}|)$$

$\mathbb{K} \to \mathbb{E}$ assigns keys to a contiguous range $\mathbb{E}$, which is then mapped to values via $\mathbb{E} \to \mathbb{V}$. The contiguity of $\mathbb{E}$ enables the values of $\mathbb{E} \to \mathbb{V}$ to be efficiently stored in an array-like data structure. We refer to the mapping $\mathbb{K} \to \mathbb{E}$ as the *enumeration* of $\mathbb{K}$, and $\mathbb{E} \to \mathbb{V}$ as the *enumerated* collection.

In this section we describe our implementation of automatic data enumeration. §III-A describes our extensions to MEMOIR [12]. §III-B describes how we transparently transform a program with automatic data enumeration. We describe optimizations to eliminate redundancy (§III-C) and transformations that uncover latent redundancy (§III-D and III-E). Next, we explain how ADE handles function calls (§III-F) and nested collections (§III-G). Then, §III-H presents the collection implementations explored in our system, including details on the specialized implementations enabled by data enumeration. Finally, §III-I introduces optimization directives to enable performance engineering with ADE.

A. Intermediate Representation

Our compiler operates on MEMOIR [12]: an extension of the LLVM IR [21] with first-class representation for data collections in an SSA form. MEMOIR extends the LLVM IR with types and operations for common data collections, i.e., sequence, map, set and tuple. The MEMOIR syntax can be found in Figures 1 and 2. MEMOIR is generated from C, C++ or Rust through a custom library that preserves information about collections instead of lowering data collections to an in-memory representation. ADE transforms collections in the program before MEMOIR is lowered to LLVM IR, where collections are laid out in memory.

In this paper, we extend MEMOIR with the *for-each* loop[1], which iterates over each element in the collection, binding the key and a reference to the value.

MEMOIR uses structured control flow, where blocks do not have names, so we use an implicit ordering for ϕ functions:

- *If-else exit:* ϕ`(value_if_true, value_if_false)`
- *Loop entry:* ϕ`(initial_value, recurrent_value)`
- *Loop exit:* ϕ`(final_value)`

We introduce nested collections, e.g., `x: Map<K,Set<V>>`. Operations are performed at a specific nesting level, e.g., `insert(x, k)` inserts into the map and `insert(x[k], v)` inserts into the nested set at `k`. The result of updates, such as `insert`, is the new state of the base collection.

We extend MEMOIR to support collection selection with type annotations. For example, `Set{HashSet}<f32>` is a set implemented with a `HashSet`. Collection types can also have an empty selection, e.g., `Set{•}<f32>`, also written as `Set<f32>` for simplicity. The available selections are described in §III-H.

B. Inserting Enumerations and Translations

We now describe the framework for automatic data enumeration before delving into optimizations. Listing 1 serves as an example MEMOIR program being transformed. As a result of our example transformation, we will enumerate `%hist`.

[1]Implemented in LLVM as a higher-order function, where a function representing the body of the loop is passed as an operand.

$v \in \textit{Variables}, f \in \textit{Functions}, n \in \mathbb{N}$

```
F ::= fn T f(v:T ...): b                          functions
b ::= if v: b else: b                             if-else
    | for [v, v] in x: b                          for-each loop
    | do: b while v                               do-while loop
    | i* b                                        sequencing
i ::= v: T? = op(x...)                            instructions
op ::= new AT                                     construction
    | read | has | size                           query
    | write | insert | remove | clear             update
    | add | sub | call | ϕ | ...                  LLVM
x ::= v | x[s] | x.n                              operands
s ::= v | n | end                                 scalars
```

Fig. 1: The MEMOIR expression syntax.

$s \in \textit{Selections} \cup \bullet$

```
 T ::= AT | ST | void                              types
AT ::= CT | Tup<T...>                             aggregates
CT ::= Seq{s}<T> | Set{s}<ST> | Map{s}<ST,T>      collections
ST ::= u64 | ... | i64 | ... | f64 | f32 | ptr | idx   scalars
```

Fig. 2: The MEMOIR type syntax.

First, the enumeration `%e: Enum` is created with type `Enum = (Enc, Dec)`, where `Enc = Map<f32,idx>` maps values to unique identifiers and `Dec = Seq<f32>` provides the reverse mapping. The enumeration is populated on the fly as new values are encountered. We introduce three translation functions:

`fn idx @enc(Enum,f32)` translates a value to its identifier. Behavior is undefined if the value is not in the enumeration.

`fn f32 @dec(Enum,idx)` translates an identifier to its value. Behavior is undefined if the identifier is not in the enumeration.

`fn (Enum,idx) @add(Enum,f32)` adds a value to the enumeration, returning the updated enumeration and the new identifier. If the value is already in the enumeration, returns the input enumeration and the value's identifier.

To transform the program, the compiler must know where to perform translations. Algorithm 1 generates three sets—*ToEnc*, *ToDec* and *ToAdd*—with use sites that must be patched by translating the value with a call to the respective function.

After collecting the uses to patch, we can transform the program, shown in Listing 2. First, we transform the type of `%hist` to `Map{BitMap}<idx,u32>` (line 4). Then, a new enumeration is allocated (line 3). Finally, all use sites are patched by calling the helper function (lines 8 and 13).

A keen eye will notice that, at this point, we have not optimized the behavior of the program. Rather, we have solely added a level of indirection! In the remainder of this section, we discuss optimizations that make ADE performant.

C. Eliminating Redundant Translations

Redundant translation elimination (RTE) is the primary driver for optimizing performance with ADE. Informally, RTE leverages the properties of the `@enc`, `@dec` and `@add` functions

Listing 1: Compute histogram of a sequence.

```
1  fn void @count(%input: Seq<f32>):
2    %hist := new Map<f32,u32>()
3    for [%i, %val] in %input:
4      %hist0 := ϕ(%hist, %hist3)
5      %cond := has(%hist0, %val)
6      if %cond:
7        %freq := read(%hist0, %val)
8      else:
9        %hist1 := insert(%hist0, %val)
10     %freq0 := ϕ(%freq, 0)
11     %hist2 := ϕ(%hist0, %hist1)
12     %freq1 := add(%freq0, 1)
13     %hist3 := write(%hist2, %val, %freq1)
```

Listing 2: Compute histogram of sequence with ADE.

```
1  type Enum = ... // describe in text.
2  fn void @count(%input: Seq<f32>):
3    %e := new Enum()
4    %hist := new Map{BitMap}<f32,u32>()
5    for [%i, %val] in %input:
6      %hist0 := ϕ(%hist, %hist3)
7      %e0 := ϕ(%e, %e2)
8      %id := call @enc(%e0, %val)
9      %cond := has(%hist0, %id)
10     if %cond:
11       %freq := read(%hist0, %id)
12     else:
13       (%e1, %id) = call @add(%e0, %val)
14       %hist1 := insert(%hist0, %id)
15     %freq0 := ϕ(%freq, 0)
16     %hist2 := ϕ(%hist0, %hist1)
17     %e2 := ϕ(%e0, %e1)
18     %freq1 := add(%freq0, 1)
19     %hist3 := write(%hist2, %id, %freq1)
```

to apply rewrite rules. The first leverages the fact that `@dec` is the inverse function of `@enc`, allowing the rewrite:

```
call @enc(e, call @dec(e, x)) → x
```

Additionally, if a value needs to be decoded, then it is already in the enumeration, allowing the rewrite:

```
call @add(e, call @dec(e, x)) → (e, x)
```

Finally, since values are never removed from the enumeration and `@dec` is injective, allowing the rewrite:

```
eq(call @dec(e, x), call @dec(e, y)) → eq(x, y)
```

This analysis is described with more detail in Algorithm 2, which collects redundant translations into *TrimEnc*, *TrimDec* and *TrimAdd*. These are used to recompute *ToEnc*, *ToDec* and *ToAdd*, by subtracting the corresponding *Trim* set.

We use a static heuristic to estimate the benefit of ADE. Informally, enumeration is beneficial iff we can find redundant translations. To compute the heuristic, we perform a *what if* analysis using the results of the FINDREDUNDANT function:

$$|\textit{TrimEnc}| + |\textit{TrimDec}| + |\textit{TrimAdd}|$$

This heuristic could be extended with profile information, however this static heuristic has sufficed for our use cases.

ALGORITHM 1: Uses to patch for enumerated collection.

in: v associative collection variable
out : $ToEnc(v)$, $ToDec(v)$, $ToAdd(v)$
foreach $r \in Redefs(v)$ **do**
  foreach $u \in Uses(r)$ **do**
    switch $User(u)$ **do**
      case `op(r, k)` **where** `op` $\in$ {`read`, `has`, `remove`} **do**
        insert use of k **in** $ToEnc(v)$
      case `write(v, r, k)` **do**
        insert use of k **in** $ToEnc(v)$
      case `insert(r, k)` **do**
        insert use of k **in** $ToAdd(v)$
      case `for [k, v] in r: ...` **do**
        insert $Uses(k)$ **in** $ToDec(v)$
      case `op(r[k]..., ...)` **do** *# Nesting, see §III-G*
        insert use of k **in** $ToEnc(v)$

ALGORITHM 2: Identify redundant translations.

function FINDREDUNDANT($ToEnc, ToDec, ToAdd$)**:**
  foreach $u \in ToDec$ **do**
    if $u \in ToEnc$ **then** *# Encoding a decoded value*
      insert u **in** $TrimDec$
      insert u **in** $TrimEnc$
    else if $u \in ToAdd$ **then** *# Already enumerated*
      insert u **in** $TrimDec$
      insert u **in** $TrimAdd$
    else if $User(u)$ **is** `eq(x, y)` $\wedge$ `x` $= UsedBy(u)$ **then**
      let $w \leftarrow$ use of `y` in $User(u)$
      if $w \in ToDec$ **then** *# Comparing enumerated values*
        insert u **in** $TrimDec$
        insert w **in** $TrimDec$
  return $TrimEnc, TrimDec, TrimAdd$

ALGORITHM 3: Find candidates for enumeration sharing.

in: A set of collection allocations in the function
out : $Candidates$ set of collections to share an enumeration
 # Track the allocations that already belong to a candidate.
let $Used = \varnothing$
 # Create a candidate for each collection in the function.
foreach $a \in A$ **where** $a \notin Used \wedge a$ **isa** `Assoc<T,U>` **do**
  let $C = \{a\}$
  insert a **in** $Used$
  # Find other collections that would benefit the candidate.
  foreach $b \in A$ **where** $b \notin Used$ **do**
    if CANSHARE(b, `T`)) $\vee$ CANPROPAGATE(b, `T`) **then**
      let $B_\Sigma =$ BENEFIT(C) + BENEFIT($\{b\}$)
      let $B_\cup =$ BENEFIT($C \cup \{b\}$)
      # Benefit must be greater than the sum of its parts.
      if $B_\cup > B_\Sigma$ **then insert** b **in** C ; **insert** b **in** $Used$
  # Only emit candidates with positive benefit.
  if BENEFIT(C) > 0 **then insert** C **in** $Candidates$
 # Benefit of sharing an enumeration between collections.
function BENEFIT(C)**:**
  let $TrimEnc, TrimDec, TrimAdd =$
    $$\text{TRIM}\left(\bigcup_{c\in C} ToEnc(c), \bigcup_{c\in C} ToDec(c), \bigcup_{c\in C} ToAdd(c)\right)$$
  return $|TrimEnc| + |TrimDec| + |TrimAdd|$
 # §III-D: Key type of collection must match to share.
function CANSHARE(x, `T`)**:**
  return x **isa** `Assoc<T,V>`
 # §III-E: Element type of collection must match to propagate.
function CANPROPAGATE(x, `T`)**:**
  return x **isa** `Assoc<K,T>` $\vee$ x **isa** `Seq<T>`

D. Sharing Enumerations Between Collections

Although enumeration can benefit a single collection, it is common for multiple collections to operate on the same data. To exploit this, we *share* an enumeration between multiple enumerated collections. This amortizes the overhead of constructing the enumeration and uncovers redundant translations to eliminate. In this section, we present algorithms and heuristics to identify cases where sharing is beneficial.

We use the term *candidate* to refer to a group of collections that share an enumeration. Informally, we identify a candidate as the maximal set of collections that improve the benefit heuristic. For a collection to be in a candidate, its inclusion must make the benefit of the candidate greater than the sum of its parts. The key insight being that, if two collections are sharing data, they are likely to be constructed from one another or have their values compared. If this is the case, redundant translations will surface when computing the benefit. Algorithm 3 shows the analysis in more detail.

E. Propagating Identifiers through Collection Elements

In addition to sharing an enumeration between the keys of sets and maps, any collection can be transformed to store identifiers in its elements using *propagation*. Consider Listing 3, where we iteratively search the `%uf` map using the value read in the last step as the key in the next step.

Listing 3: Find parent in union-find *without* propagation.

```
1  fn u32 @find(%uf:Map<idx,u32>, %v:u32, %e:Enum):
2    %id_v = call @enc(%e, %v)
3    do: %curr := ϕ(%v, %parent)
4        %id_curr := ϕ(%id_v, %id_parent)
5        %e0 := ϕ(%e, %e1)
6        %parent := read(%uf, %id_curr)
7        (%e1, %id_parent) = call @add(%e0, %parent)
8        %not_done := neq(%id_parent, %id_curr)
9    while %not_done
10   %found := ϕ(%parent)
11   ret %found
```

If we naïvely apply our transformation to this code, the key is translated on each iteration, adding costly accesses to a potentially hot loop. Propagation addresses this by mutating `%uf` to store identifiers for both its keys *and* values. Listing 4 shows the same program with propagation applied to the elements of `%uf`. As a result, there is no translation in the loop. It has been replaced by a single decode outside the loop.

We refer to a collection storing identifiers in its elements as a *propagator*. Algorithm 4 details how we identify the uses to patch for propagators. Since we cannot determine that a given value is an identifier or not at runtime, propagators must

Listing 4: Find parent in union-find *with* propagation.

```
fn u32 @find(%uf:Map<idx,idx>, %v:u32, %e:Enum):
  %id_v = call @enc(%p, %v)
  do: %id_curr := ϕ(%id_v, %id_parent)
      %id_parent := read(%uf, %id_curr)
      %not_done := neq(%id_parent, %id_curr)
  while %not_done
  %id_found := ϕ(%id_parent)
  %found := call @dec(%p, %id_found)
  ret %found
```

ALGORITHM 4: Uses to patch for propagator.

in: v a collection variable
out : $ToEnc(v)$, $ToDec(v)$, $ToAdd(v)$ uses to patch.
foreach $r \in Redefs(v)$ **do**
  foreach $u \in Uses(r)$ **do**
    switch $User(u)$ **do**
      case `v := read(r, k)` **do**
        insert $Uses(v)$ **in** $ToDec(v)$
      case `write(v, r, k)` **do**
        insert use of k **in** $ToAdd(v)$
      case `for [k, v] in r: ...` **do**
        insert $Uses(v)$ **in** $ToDec(v)$

store identifiers in *all* elements. As such, all values written to the collection must be added to the enumeration. Algorithm 3 shows how we apply the benefit heuristic to identify beneficial propagators for a candidate. We employ the same heuristic for propagation and sharing, as the key insight holds for both.

F. Interprocedural Transformation

To handle enumerated collections passed as arguments, the compiler determine which enumeration to use for translation in the callee, detailed in Algorithm 5. This generates a set of equivalence classes, where items in a class share the same enumeration. Each class is given a global variable to store the enumeration. When patching a use, the enumeration is loaded from this global variable for the translation.

For recursive functions, ADE reuses the enumeration at each invocation, avoiding construction overhead. Without reuse, the constructing an enumeration at each invocation led to timeouts and significant memory overhead for our benchmarks.

For functions that are externally visible, or have parameters that are only enumerated for some callers, we create a clone of the function to transform. We do not apply ADE to collections passed into calls with indirect or externally defined callees, nor collections that escape into unknown memory locations. This ensures correct translation for all uses and functionally prevents transformation of collections shared among parallel threads, but permits correct handling of sequential programs and thread-local collections.

G. Enumerating Nested Collections

We extend ADE to operate on nested collections by making our analyses conscious of nesting. We refer to a nested

ALGORITHM 5: Unify collections that share an enumeration.

in: V the set of all candidates
in: F the set of functions with enumerated parameters
out : $EnumClass$: equivalence classes for each enumeration
 # Unify redefinitions.
let U *be a union-find data structure*
foreach $r \in Redefs(v)$ **where** $v \in V$ **do**
  unify v **with** r **in** U
 # Unify parameters, see Figure 3 for a visual aide.
foreach $p \in Params(f)$ **where** $f \in F$ **do**
  # Unify parameters where all arguments agree.
  foreach $a \in Incoming(p)$ **where** $U.find(a) = b$ **do**
    unify p **with** b **in** U
  # Unify parameters whose arguments are unified at each call.
  if $\exists q \in Params(f)$ **where** $\forall c \in Callers(f)$, $U.find(c.arg(p)) = U.find(c.arg(q))$ **then**
    unify p **with** q **in** U
 # Compute the equivalence classes from the union-find.
let $EnumClass$ = **classes**(U)

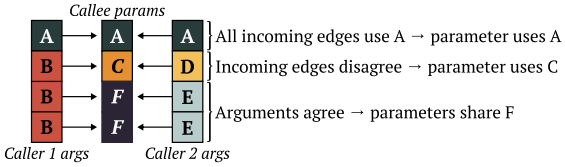


Fig. 3: Visualization of unifying parameters in Algorithm 5.

collection using the operand syntax from Figure 1. If a nested collection is enumerated, all collections at that nesting level share an enumeration. For example, given `%x:Seq<Set<f32>>`, `%x[0]` and `%x[1]` use the same enumeration. To ensure that all keys are properly translated, we patch nested operand indices as shown in the last **case** statement of Algorithm 1.

H. Collection Selection

ADE makes specialized implementations possible by manufacturing the necessary properties, expanding the selection space. Table I outlines the implementations available to each type, where the **Enum'd?** column marks specialized implementations for enumerated collections. The rest of this section describes those specialized implementations in more detail.

`BitSet` is implemented with a `boost::dynamic_bitset`, a contiguous array of bits that supports dynamic resizing. Dynamic resizing is necessary for our application since enumerations are constructed on the fly. `BitMap` is implemented similarly, with a contiguous array to hold the values.

`SparseBitSet` is implemented with Roaring [22], a popular compressed bitset library used by Apache Lucene [23] and Youtube [24], and studied by academia [25]. Roaring uses a hybrid compression technique, combining uncompressed bitsets, packed arrays and run-length encoding to achieve a performant yet compact implementation.

TABLE I: Collection selection candidates and the average case complexity of basic operations, where n is the number of elements in the collection k is the largest integer key in the collection.

Type	Enum'd?	Selection	Description	Rd/Wr	Insert	Remove	Storage
`Seq<T>`	✗	`Array`	Resizeable array	1	$O(n)$	$O(n)$	$n \cdot \textbf{bits}(T)$
`Set<T>`	✗	`HashSet`	Hash table	—	$O(1)$	$O(1)$	$O(n \cdot \textbf{bits}(T))$
	✗	`FlatSet`	Sorted array	—	$\log(n)$	$\log(n)$	$n \cdot \textbf{bits}(T)$
⋮	✗	`SwissSet`	Swiss table	—	$O(1)$	$O(1)$	$O(n \cdot (1 + \textbf{bits}(T)))$
	✓	`BitSet`	See §III-H	—	1	1	k
	✓	`SparseBitSet`	See §III-H	—	$O(1)$	$O(k)$	$O(k)$
`Map<K,T>`	✗	`HashMap`	Hash table	$O(1)$	$O(1)$	$O(1)$	$O(n \cdot (\textbf{bits}(K) + \textbf{bits}(T))$
	✗	`SwissMap`	Swiss table	$O(1)$	$O(1)$	$O(1)$	$O(n \cdot (1 + \textbf{bits}(K) + \textbf{bits}(T))$
⋮	✓	`BitMap`	See §III-H	1	1	1	$k \cdot (1 + \textbf{bits}(T))$

I. Performance Engineering with Optimization Directives

Modern compilers are becoming open boxes [26–28], where performance engineers are given the ability to dictate optimization plans to the compiler. In this vein, we provide directives to specify candidates and constraints that override the benefit heuristic. This allows performance engineers to fine-tune ADE, while still benefiting from fully automatic transformation.

Examples of these directive are shown in Listing 5. Users can explicitly enable (line 1) or disable (line 3) data enumeration for a collection. Additionally, they can specify that a collection should not share an enumeration with another (line 1). Alternatively, they can specify a group of collections to share an enumeration between (lines 5 and 7). Finally, users can select the implementation for a given collection (line 3).

Listing 5: Directives allow users to control and guide ADE.

```
1 #pragma ade enumerate noshare(c)
2 a = new Set<u32>
3 #pragma ade noenumerate select(SwissMap)
4 b = new Map<u32, u32>
5 #pragma ade share group("d+e group")
6 c = new Set<u32>
7 #pragma ade share group("d+e group")
8 d = new Set<u32>
```

IV. Evaluation

We answer the following research questions:

- RQ1 What is the impact of ADE on performance and memory usage, and are there differences across architectures?
- RQ2 What proportion of sparse accesses are eliminated?
- RQ3 What is the impact of our techniques to uncover and eliminate redundant translations?
- RQ4 Can user directives enable further benefits?
- RQ5 How does ADE compare to state-of-the-art collection implementations from industry?

A. Experimental Setup

We perform our evaluation on two machines:

Intel-x64: 176 core Intel Xeon Gold 6238L CPU at 2.10GHz with 2.8MiB 8-way L1d$, 88MiB 16-way L2$, 121MiB 11-way LL$, and 376GiB of DDR4 at 3200MT/s.

AArch64: 128 core ARM Neoverse N1 at 3GHz with 8MiB 4-way L1d$, 8MiB 4-way L2$, 128MiB 8-way LL$, and 256GiB of DDR4 at 3200MT/s.

Both machines run Ubuntu 22.04.1 LTS. Our compiler is built upon the MemOIR compiler [12] using LLVM 18.1.8 [21], with abstractions from NOELLE [29].

We implement 15 of the 17 Lonestar 'Analytics' benchmarks [30] and `freqmine` from PARSEC [31]. These benchmarks are implemented in C++ using abstract MemOIR data collection types instead of low-level implementations, representing code written by developers before heavy manual optimization. The benchmarks are listed in Figure 4, with the breakdown of collection operations executed in each, and a hierarchical clustering of benchmarks based on the breakdown. We use suite inputs where provided, otherwise we use an input from SNAP [32, 33]. All benchmarks are compiled with full LTO and `-O3`. Results shown are the median of 10 runs, with the geometric mean (GEO) reported in each plot.

RQ1: Performance

We compare ADE against the MemOIR compiler on Intel-x64. The only difference between these two compilers are the analyses and transformations detailed in §III. We present speedup for the whole-program (Figure 5a) and region of interest, where initialization is factored out (Figure 5b).

We find that ADE provides substantial whole-program speedups of 2.12× on average, up to 8.72×, with a sole regression on KC of 0.94×. KC is a case where the cost of constructing the enumeration (1.16× initialization slowdown) is not amortized by the ROI speedup (2.01×) since > 90% of execution time is initialization. Focusing solely on the region of interest, we see speedups of 2.98× on average, up to 9.02×. These results show that ADE optimizations and specialized implementations overcome the overhead of enumeration construction and translation for significant performance gains.

We evaluate memory usage using maximum resident size and present our findings in Figure 5c. We find that ADE reduces maximum resident size by 5.6% on average, up to 50.7%. However, we see no impact on the majority of benchmarks. We see one regression on FIM with a 27.3% increase. The regression is caused by our static benefit heuristic, which results in the compiler enumerating cold collections that are only used during verbose output. Since verbose output is disabled for the PARSEC input, the additional mappings are allocated and never used. While ADE does not consistently reduce memory usage,

Abbr.	Benchmark
PTA	LLVM Points-To Analysis
FIM	Frequent Itemset Mining
MST	Minimum Spanning Tree
BP	Bipartion
CC	Connected Components
PP	Preflow-Push
CD	Community Detection
PR	PageRank
BFS	Breadth-first Search
KC	k-Core
KT	k-Truss
TC	Triangle Counting
IS	Independent Set
MCBM	Max Card. Bipartite Matching
SSSP	Single-Source Shortest Path
BC	Betweenness Centrality

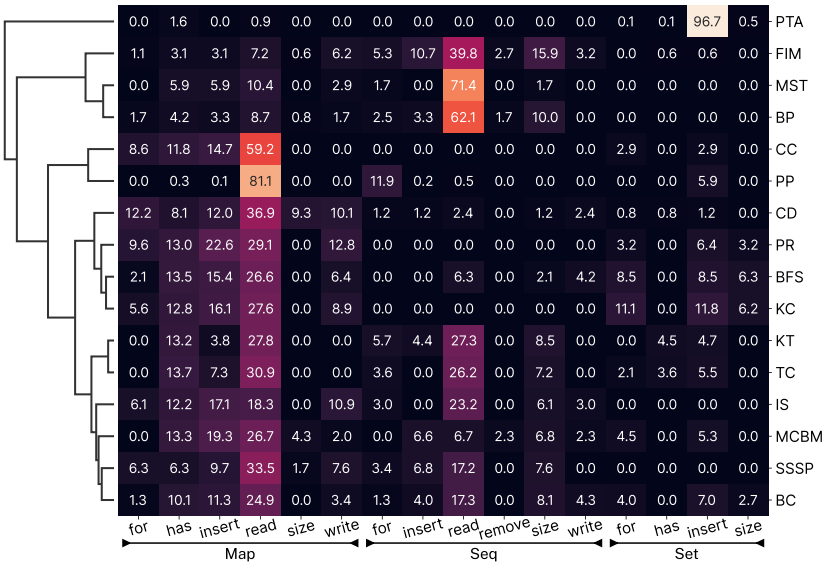
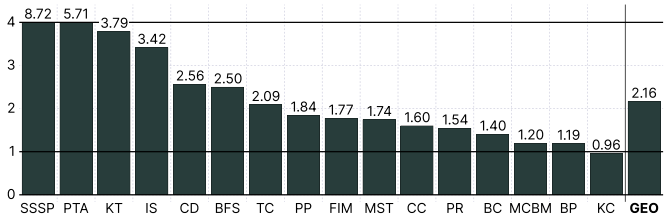
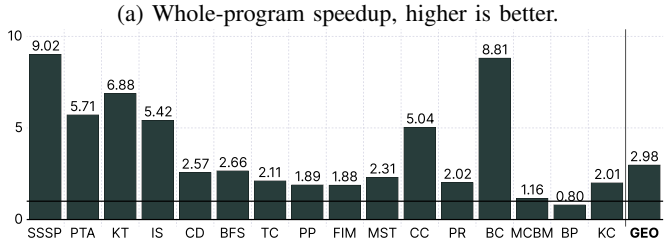


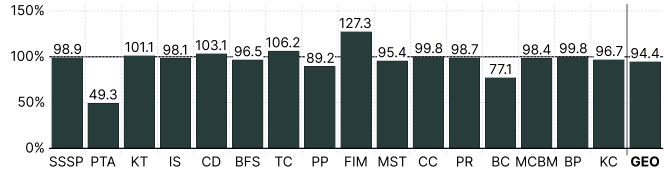
Fig. 4: List of benchmarks, with hierarchical clustering based on the breakdown of dynamic collection operations executed.



(a) Whole-program speedup, higher is better.

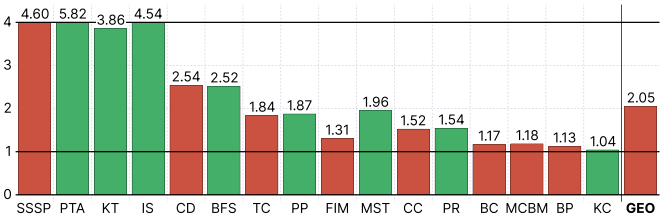


(b) ROI speedup, higher is better.

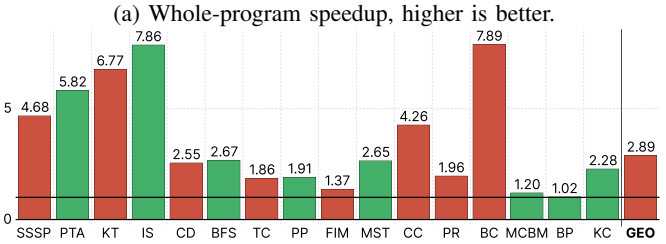


(c) Maximum resident set size, lower is better.

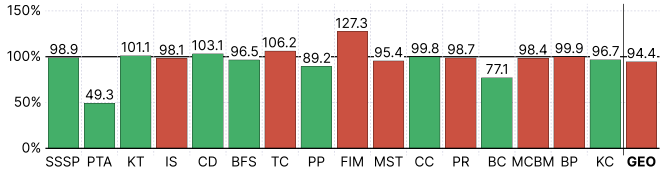
Fig. 5: Performance and memory usage of ADE compared to MEMOIR running on Intel-x64.



(a) Whole-program speedup, higher is better.



(b) ROI speedup, higher is better.



(c) Memory usage, lower is better.

Fig. 6: Performance and memory usage of ADE relative to MEMOIR on AArch64.

it achieves the aforementioned performance results without bloated memory usage.

With recent shifts in server and consumer computing markets from x86-64 towards AArch64, it is important to examine the impact of compiler optimizations across both architectures. Figure 6 shows the performance results of ADE on AArch64. We found no significant difference in relative memory usage between Intel-x64 and AArch64. Bars are shaded green if the results are better on AArch64 than Intel-x64, and red if worse.

We find that ADE speedups are robust across architectures, with whole-program speedups of 2.03× on average, up to 11.35×, and ROI speedups of 2.91× on average, up to 7.89×. We find some benchmarks with significant regression on AArch64, such as SSSP, where whole-program speedup falls from 8.72× to 4.60×. This is caused by operation-level performance differences across architectures (Table III). Specifically,

TABLE II: Sparse/dense access counts relative to MEMOIR.

	MEMOIR		ADE		Difference		
Bench	*Sparse*	*Dense*	*Sparse*	*Dense*	*Sparse*	*Dense*	*Total*
BC	48.8	51.2	40.1	110.6	-8.7	**+59.4**	**+50.7**
BFS	100.0	0.0	3.2	98.2	-96.8	**+98.2**	**+1.4**
BP	6.3	93.7	6.2	93.5	-0.1	**-0.2**	**-0.3**
CC	100.0	0.0	32.0	101.0	-68.0	**+101.0**	**+33.0**
CD	100.0	0.0	39.8	60.3	-60.2	**+60.3**	**+0.0**
FIM	33.0	67.0	5.6	102.2	-27.4	**+35.2**	**+7.7**
IS	64.9	35.1	5.7	117.0	-59.2	**+81.9**	**+22.7**
KC	100.0	0.0	151.0	82.1	+51.0	**+82.1**	**+133.0**
KT	53.0	47.0	5.3	102.2	-47.7	**+55.2**	**+7.4**
MCBM	74.1	25.9	63.7	115.6	-10.4	**+89.7**	**+79.3**
MST	5.0	95.0	2.7	100.2	-2.3	**+5.2**	**+2.9**
PP	99.3	0.7	8.7	100.2	-90.5	**+99.4**	**+8.9**
PR	100.0	0.0	24.2	95.7	-75.8	**+95.7**	**+19.9**
PTA	100.0	0.0	30.4	107.6	-69.5	**+107.5**	**+38.0**
SSSP	50.0	50.0	0.1	100.0	-50.0	**+50.0**	**+0.1**
TC	19.7	80.3	0.1	380.3	-19.6	**+300.0**	**+280.4**

TABLE III: Per-operation speedup of different implementations relative to `Hash{Set,Map}`.

	Impl.	Read	Write	Insert	Remove	Iterate	Union
Intel-x64	BitSet	—	—	9.08	1.24	0.19	5817.38
	SparseBitSet	—	—	1.54	1.07	0.27	3700.50
	SwissSet	—	—	1.61	0.40	0.27	1.71
	FlatSet	—	—	0.19	0.10	5.59	25.31
	BitMap	10.63	15.94	13.10	1.32	2.65	—
	SwissMap	0.69	1.46	2.58	0.41	3.65	—
AArch64	BitSet	—	—	12.53	2.63	0.22	6944.48
	SparseBitSet	—	—	2.81	2.21	0.29	4702.13
	SwissSet	—	—	1.46	0.52	0.28	3.28
	FlatSet	—	—	0.28	0.22	3.15	50.37
	BitMap	18.65	10.20	8.91	2.60	6.41	—
	SwissMap	0.64	0.65	1.18	0.51	7.16	—

writes and insertions to a `BitMap`, hot operations for SSSP, see 1.56× and 1.47× slowdown on AArch64, respectively.

RQ2: Sparse and Dense Accesses

In §III we described the high-level goal of ADE: replacing sparse accesses with dense accesses. We evaluate the composition of these accesses in our benchmarks, shown in Table II, and find that ADE effectively replaces sparse accesses with dense accesses. We find that ADE may increase the total number of dynamic accesses, however performance results suggest that this is a beneficial tradeoff.

To understand this tradeoff, we measure relative performance of different implementations across machines, shown in Table III. We find that all operations, aside from set iteration, are significantly more performant on a `Bit{Set,Map}` compared to both `Hash{Set,Map}` and `Swiss{Set,Map}`.

RQ3: Ablation Study

In §III, we introduced a battery of optimizations to uncover and eliminate redundant translations. We perform an ablation study to understand the impact of each. The results are shown in Figures 7 and 8 relative to ADE with all optimizations.

First, we evaluate the impact of redundant translation elimination (RTE), as described in §III-C. The results of disabling RTE are seen in Figure 7a. We find that RTE is beneficial across the board, with an average 2.63× slowdown when disabled, up to 16.7×. We also find that RTE has no impact on memory usage, which is expected because it solely removes redundant access operations.

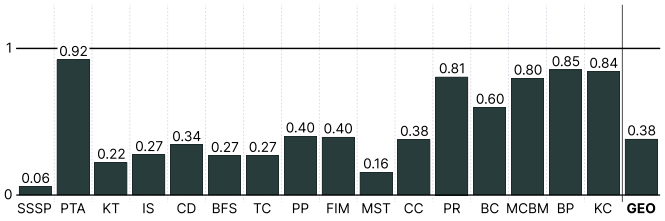
Next, we evaluate propagation (§III-E), shown in Figure 7b. These results show the primary benefit of propagation: uncovering latent redundancy, where elements are used to primarily transfer values from one enumerated collection to another.. This is seen in the correlation between RTE and propagation ablation speedups. In cases, such as SSSP, this is especially prevalent as we see approximately equivalent slowdowns when either RTE or propagation are disabled. This indicates that, in such cases, the compiler is able to uncover little to no redundancy without propagation. We find that propagation has no impact on memory usage for the majority of benchmarks, with the exception of FIM where we see memory usage balloon to 1694.3% This is due to propagation being necessary to enable sharing, without which we introduce many enumeration mappings. This is supported by the memory usage seen with sharing disabled, which similarly grows to 1696.1%

Finally, we evaluate sharing (§III-D), shown in Figures 7c and 8. Of note, no sharing also entails no propagation, as the benefit computation will never introduce a propagator unless it can be shared with an enumerated collection. Similarly to propagation, sharing uncovers latent redundancy, which is illustrated by a trend towards the slowdowns seen with RTE disabled. Unlike the prior optimizations, we find that sharing has a significant impact on memory usage, rising by an averge of 20% when disabled. However, we see one outlier in PTA, which achieves significant speedup and memory usage reduction with sharing disabled. This is caused by our benefit heuristic being overly eager to share an enumeration between collections, leading to `Bit{Set,Map}` that are sparse over the shared enumeration. We use this outlier as a case study for our user-guided optimization directives in RQ4.

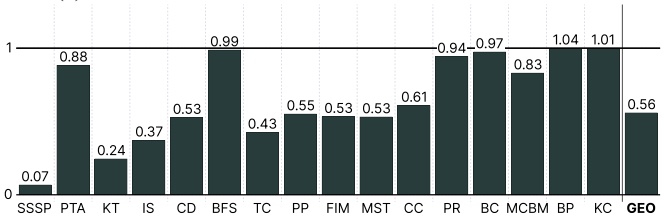
RQ4: Performance Engineering Case Study

In RQ3, we found that ADE heuristics were too optimistic for sharing enumerations in PTA. We use PTA as a performance engineering case study, using the directives from §III-I.

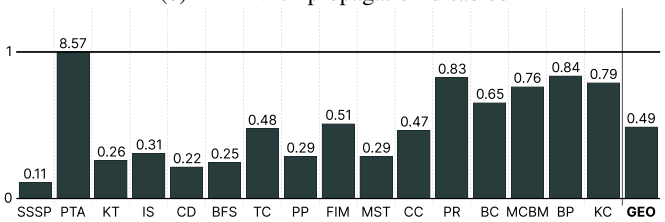
We first bisect the problem by disabling sharing for each collection in the program using the `noshare` directive. Doing so, we find that the regression was caused by sharing with the points-to set (`Map<ptr,Set<ptr>> %pts`), particularly the inner sets `%pts[*]`. By disabling sharing for the inner sets, we achieve a speedup of 78.1× and reduce memory usage by 71.0% relative to MEMOIR (13.7× speedup and 41.1% memory usage reduction relative to untuned ADE). The root cause of this speedup is that the inner sets range over all objects, while the original enumeration ranges over all pointers. There are approximately 1.8×10^3 allocations and 2×10^7 pointers



(a) ADE with redundant translation elimination disabled



(b) ADE with propagation disabled



(c) ADE with sharing disabled

Fig. 7: Ablation study, whole-program speedup with optimization techniques disabled on Intel-x64 relative to ADE with all optimizations enabled.

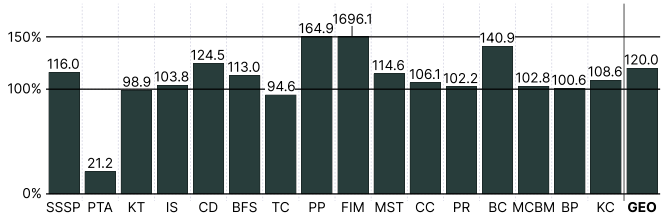


Fig. 8: Ablation study, memory usage with sharing disabled on Intel-x64 relative to ADE with all optimizations enabled. RTE had no impact on memory usage, while propagation only impacted FIM with similar results to sharing.

in the `sqlite3` bitcode input we use, resulting in the inner bitsets using only 0.009% of their bits. This led to a large performance penalty for aggregate operations, such as iterating over the bitset or computing the number of set bits.

Seeing these results, we investigated if data enumeration is even beneficial for the inner sets with the `noenumerate` directive, but this resulted in a speedup of only 1.12× and a memory usage reduction of 25.9% relative to MEMOIR.

Finally, we investigate alternative implementations for the inner set using the `select` directive. We first selected a `SparseBitSet` to compact the memory layout by eliminating unused bits. This resulted in a 1.57× speedup and 67.1% reduction in memory usage compared to untuned `BitSet`. Next, we selected a `FlatSet` so that only items in the set are stored, with linear set union (a hot operation). This resulted in a 2.73× speedup and 20.4% reduction in memory usage compared to `BitSet`. While these results showed a good improvement, they fall short of the 78.1× speedup and 71.0% reduced memory usage achieved with `noshare`.

RQ5: Comparison with Third-Party Implementations

While the C++ standard template library is commonly used by developers, it has known performance limitations. Because of this, developers make use of third party libraries, such as Abseil [34]. We compare ADE against Abseil's swiss table, a highly optimized implementation of sets and maps. We present our findings in Figures 9 and 10 for Intel-x64. We did not see many significant differences on AArch64, we will discuss the differences we did find later in this section.

First, we directly compare `Swiss{Set,Map}` against `Hash{Set,Map}` by setting the default implementation of the MEMOIR compiler. We present the results in Figures 9 and 10. On average, we find that `Swiss{Set,Map}` is more performant and uses less memory. However, we see major performance regressions on SSSP, KT, PP and FIM. These regressions are in stark contrast to large gains on PR and MCBM, with speedups of 3.43× and 2.69×, respectively.

Using the direct comparison as a backdrop, we compare ADE against MEMOIR, where both use `Swiss{Set,Map}` as the default implementation. Performance results are shown in Figure 9b. We find that most benchmarks have similar speedup results as in RQ1. This is with the sole exception of MCBM, where `Swiss{Set,Map}` provide a better solution than ADE in this use case, but tuning with directives could help.

Looking at memory usage in Figure 10b, we find that ADE provides a significant improvement over `Swiss{Set,Map}` in PTA and TC, where memory usage is reduced by 60.0% and 70.4%, respectively, with a sole regression on FIM of 27.6%.

On AArch64, we saw two deviations from the Intel-x64: SSSP saw performance regressions using `Swiss{Set,Map}`, while IS saw performance improvements using `Swiss{Set,Map}`.

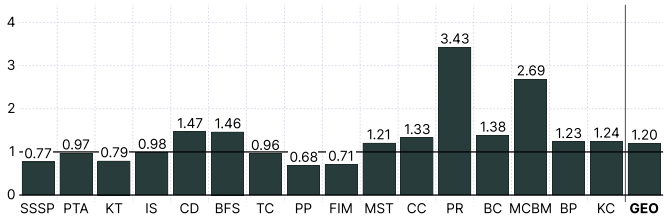
V. Related Works and Opportunities

ADE differs from prior approaches to collection selection by *enabling specialized collections through program transformation*. While prior works present a variety of techniques for *exploring* the selection space, none *expand* it. This makes ADE a fundamentally new approach to memory-centric optimization, where program transformation and collection selection can be combined to achieve more than the the sum of their parts. This section discusses prior work on collection selection (§V-A) and memory-centric transformations (§V-B) with the goal of providing both a survey and identifying avenues of future work using ADE as a signpost.

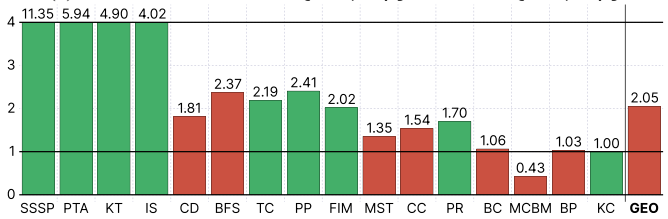
A. Collection Selection

We classify prior work on collection selection into three categories: *runtime*, *search-based*, and *synthesis*.

Runtime selection techniques fully embrace the dynamicity of collection selection, using profiling to select the

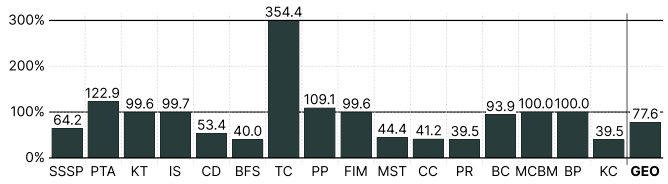


(a) MEMOIR with `Swiss{Set,Map}` over `Hash{Set,Map}`.

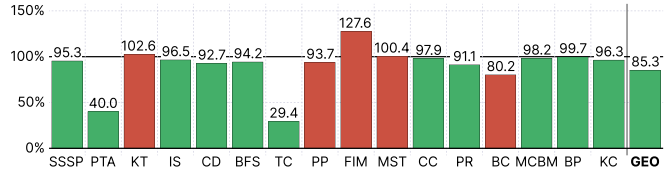


(b) ADE over MEMOIR with `Swiss{Set,Map}`.

Fig. 9: Whole-program speedup of MEMOIR and ADE with and against `Swiss{Set,Map}` on Intel-x64, higher is better.



(a) MEMOIR with `Swiss{Set,Map}` relative to `Hash{Set,Map}`.



(b) ADE with `Swiss{Set,Map}` relative to `Swiss{Set,Map}`.

Fig. 10: Maximum resident set size comparison with `Swiss{Set,Map}` on Intel-x64, lower is better.

implementation. CoCo [10] uses a complexity-guided runtime selection solution for Java using 'collection combos', a set of implementations with a shared interface. One implementation in the combo is active at a time, while inactive collections are updated to hold abstract elements that can be used to compute the concrete value when the selection changes. Seeing the overhead of instance-level adaptation, CollectionSwitch [4] coarsens runtime selection to the allocation site. By combining runtime selection with just-in-time transformation to introduce desired properties, specialized implementations could be enabled. Additionally, a runtime system could inspect collections for such desired properties.

Search-based selection relies on runtime profiles to select implementations for hot operations. `perflint` [2] first used this strategy to identify inefficient collection usage in C++, recommending source code changes to fix them, e.g., converting `set` to `unordered_set`. Brainy [3] extended this with a machine-learned model that is more effective across inputs. Similar ideas have since been explored in both Java and C++ [6–8] and could be use ADE as a selection target. Collection Skeletons [5] takes a different approach, searching for an implementation that satisfies the user-specified properties of the collection, e.g., uniqueness, which could be enhanced with manufactured properties.

Data structure synthesis takes a data structure description and synthesizes an implementation. Many approaches [35, 36] use a relational model of the program's data for synthesis, with extensions to support concurrency [37]. CRES [11] extends these techniques to use static analysis instead of runtime models, eliding profiling cost. Furthermore, Cozy [9] enables users to express ad-hoc properties via query descriptions, such as the directedness or cyclicity of a graph, which are leveraged during synthesis. To build upon this, MEMOIR collections can be described to these systems as relations, and specialized implementations can be synthesized using a toolbox of transformations that manufacture properties.

B. Automatic Memory Optimizations

Performance engineers employ a variety of memory optimization techniques, from bespoke pointer representations [38] to hand-rolled garbage collection [39]. However, most of these techniques demand considerable manual effort with little to no automation. To reduce this burden, prior work has shown that compilers can automate several memory optimizations, such as pool allocation [40], pointer compression [41], array interleaving [42], memory defragmentation [39] and data structure splicing [12, 43, 44]. This paper adds data enumeration to this list, however the directions still remains largely unexplored.

VI. CONCLUSIONS

In this paper, we have shown that compilers are capable of automating data enumeration, achieving up to 78.1× speedup and 71.0% reduced memory usage without the need for tedious manual effort. ADE illustrates how compilers can manufacture data properties to enable general-purpose programs to benefit from specialized collection implementations, making it the first approach to enhance collection selection with co-designed transformations. While this work focuses on data enumeration for sets and maps, we see clear opportunities for extending these ideas to additional properties and collection types through integration with prior work. ADE also demonstrates the power of collection-oriented compiler infrastructures, like MEMOIR [12], which make the necessary analysis and transformation possible. We believe that such infrastructure is essential for future work in this direction, and that our results establish a new envelope for collection-centric compiler optimizations, paving the way for compilers that treat collections as first-class optimization targets.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grants NSF-2119069, NSF-2148177 and NSF-2107042. All opinions, findings, conclusions, and recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the NSF.

ARTIFACT APPENDIX

A. Abstract

Our artifact includes source files for the MEMOIR compiler, with the transformation and extensions described in the paper. In addition to this, it includes the benchmark suite used for evaluation, and plotting scripts. This artifact reports its results by recreating Figures 4, 5, 6, 8 and 9 of the paper. The user is able to configure which experiments they would like to evaluate, this customization is detailed in the artifact appendix. The artifact has been tested on both Intel-x64, AMD-x64, and AArch64. The artifact requires a network connection to download external dependencies and our benchmark suite.

B. Artifact Check-list (meta-information)

- **Algorithm: Automatic Data Enumeration**
- **Program: MEMOIR**
- **Compilation: LLVM18.1.8**
- **Transformations: Automatic Data Enumeration**
- **Binary: None**
- **Data set: SNAP, Lonestar and PARSEC**
- **Run-time environment: Linux**
- **Hardware: x64**
- **Run-time state: Yes**
- **Execution: Sole-user**
- **Metrics: Execution time, Maximum resident set size**
- **Output: Figure 4; Optional: Figures 5, 6, 8, 9**
- **Experiments: Performance Evaluation**
- **How much disk space required (approximately)?: 15 GiB**
- **How much time is needed to prepare workflow (approximately)?: 10 minutes**
- **How much time is needed to complete experiments (approximately)?: 2 hours (1 run, with minimal configurations enabled), 10 hours (1 run, with all configurations enabled)**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: MIT License**
- **Data licenses (if publicly available)?: N/A**
- **Workflow framework used?: Docker, Unix Makefiles, Bash, customization described below.**
- **Archived?: Yes, https://doi.org/10.5281/zenodo.17633687**

C. Description

1) How delivered: The artifact is available on Zenodo [45].

2) Hardware dependencies: An x64 processor. This artifact has been tested on AMD EPYC 7443P, Intel Xeon Gold 6238L, and ARM Neoverse N1.

3) Software dependencies: The artifact depends on Docker, all other dependencies are installed during setup. This artifact has been tested with both Docker 27.5.1 and Podman 4.9.4.

4) Data sets: All inputs provided, generated or downloaded during setup, sourced from SNAP, Lonestar and PARSEC.

D. Installation

Docker will install all of the required dependencies during setup. To setup and run the docker image, run:

```
make docker
```

If you are using a container tool other than `docker`, run:

```
make docker DOCKER=/path/to/tool
```

E. Experiment workflow

First, configure the experiment workflow:

```
make config
```

After configuration, run the experiments:

```
make all
```

This will run the following workflow:

1) MEMOIR is compiled (`make compiler`).
2) Inputs are generated, unzipped or downloaded (`make inputs`).
3) All benchmarks are executed, and statistics about their execution time and memory usage are gathered via `/usr/bin/time` (`make run`).
4) All (configured) figures are created (`make plot`).

F. Evaluation and expected result

When the experiment workflow is complete, the figures will be placed in `bench/data/<TIMESTAMP>`.

Expected: We expect all figures to approximate the trend seen in the paper. However, due to differences between library implementations across machines, expect some variance.

Key Results.

`Figure-4a.pdf`: whole-program speedup of ADE over MEMOIR. *Expected:* ~2× geomean.

`Figure-4b.pdf`: region-of-interest (ROI) speedup of ADE over MEMOIR. *Expected:* ~3× geomean.

`Figure-4c.pdf`: memory usage of ADE relative to MEMOIR. *Expected:* ~100% geomean, <200% maximum. *Note:* We have seen variance in the PTA memory usage result across machines.

(Optional) **Ablation Study Results.** *Note:* We have seen variance in the PTA result across machines, this is caused by differences in the C++ STL implementation.

`Figure-5a.pdf`: speedup, redundant translation elimination disabled.

`Figure-5b.pdf`: speedup, propagation disabled.

`Figure-5c.pdf`: speedup, sharing disabled.

`Figure-6.pdf`: memory usage, sharing disabled.

(Optional) **Abseil Results.**

`Figure-8a.pdf`: speedup of MEMOIR with `Swiss{Set,Map}` over MEMOIR with `Hash{Set,Map}`.

`Figure-8b.pdf`: speedup of ADE with `Hash{Set,Map}` over MEMOIR with `Swiss{Set,Map}`.

`Figure-8c.pdf`: speedup of ADE with `Swiss{Set,Map}` over MEMOIR with `Swiss{Set,Map}`.

`Figure-9a.pdf`: memory usage of MEMOIR with `Swiss{Set,Map}` relative to MEMOIR with `Hash{Set,Map}`.

`Figure-9b.pdf`: memory usage of ADE with `Hash{Set,Map}` relative to MEMOIR with `Swiss{Set,Map}`.

`Figure-9c.pdf`: memory usage of ADE with `Swiss{Set,Map}` relative to MEMOIR with `Swiss{Set,Map}`.

G. Experiment customization

We provide various knobs for any evaluator to customize how they would like to run the experiment. An evaluator can customize their configuration by running `make config` in the top-level directory of the artifact. This configuration can be manually set in the `Makefile.config` file. The configuration options are as follows:

- **Number of Trials** can be set with the `TRIALS` environment variable. This will run each configuration of each benchmark `TRIALS` times. By default, `TRIALS=3`.
  Note: This will increase the evaluation time by a multiplication of `TRIALS`.
- **Select Benchmark** with the `BENCH` environment variable. If set, this will run a *single* benchmark instead of the full suite. This uses the name of the benchmark directory in `bench/benchmarks/<NAME>`, not the abbreviation. By default, this value is unset so that all benchmarks run.
- **Select Configurations** with the `CONFIGS` environment variable, a space-separated string of configuration names. By default, `CONFIGS="memoir ade"`. The available configurations are:
  - `memoir`: MEMOIR compiler (baseline)
  - `ade`: ADE optimization
  - `memoir-abseil`: MEMOIR compiler, using Abseil `Swiss{Set,Map}`
  - `ade-abseil`: ADE, using Abseil `Swiss{Set,Map}`
  - `ade-noredundant`: MEMOIR compiler with ADE, redundant translation elimination (§III-C) disabled
  - `ade-nopropagation`: MEMOIR compiler with ADE, propagation (§III-E) disabled
  - `ade-nosharing`: MEMOIR compiler with ADE, sharing (§III-D) disabled
  - `ade-sparse`: ADE, using `SparseBitSet`
  - `ade-nested-sparse`: ADE, using `SparseBitSet`, instead of `BitSet`, for sets (requires `BENCH=points_to`)
  - `ade-inner-no-share`: ADE, using `SparseBitSet`, instead of `BitSet`, for nested sets (requires `BENCH=points_to`)

H. Notes

For more detailed information about the artifact and how to extend it to be used on *new benchmarks and compiler configurations*, see the `README.md` in the artifact.

I. Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging

REFERENCES

[1] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle, "Energy profiles of java collections classes," in *Proceedings of the 38th International Conference on Software Engineering*, 2016. [Online]. Available: https://doi.org/10.1145/2884781.2884869

[2] L. Liu and S. Rus, "Perflint: A context sensitive performance advisor for c++ programs," in *CGO*, 2009. [Online]. Available: https://doi.org/10.1109/CGO.2009.36

[3] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande, "Brainy: effective selection of data structures," in *PLDI*, 2011. [Online]. Available: https://doi.org/10.1145/1993498.1993509

[4] D. Costa and A. Andrzejak, "Collectionswitch: a framework for efficient and dynamic collection selection," in *CGO*, 2018. [Online]. Available: https://doi.org/10.1145/3168825

[5] B. Franke, Z. Li, M. Morton, and M. Steuwer, "Collection skeletons: Declarative abstractions for data collections," in *SLE*, 2022. [Online]. Available: https://doi.org/10.1145/3567512.3567528

[6] O. Shacham, M. Vechev, and E. Yahav, "Chameleon: adaptive selection of collections," *ACM SIGPLAN Notices*, 2009. [Online]. Available: https://doi.org/10.1145/1543135.1542522

[7] M. Basios, L. Li, F. Wu, L. Kanthan, and E. T. Barr, "Darwinian data structure selection," in *ESEC/FSE*, 2018. [Online]. Available: https://doi.org/10.1145/3236024.3236043

[8] N. Couderc, E. Söderberg, and C. Reichenbach, "Jbrainy: Micro-benchmarking java collections with interference," in *Companion of the ACM/SPEC International Conference on Performance Engineering*, 2020. [Online]. Available: https://doi.org/10.1145/3375555.3383760

[9] C. Loncaric, E. Torlak, and M. D. Ernst, "Fast synthesis of fast collections," in *PLDI*, 2016. [Online]. Available: https://dl.acm.org/doi/10.1145/2908080.2908122

[10] G. Xu, "Coco: Sound and adaptive replacement of java collections," in *ECOOP 2013*, 2013.

[11] C. Wang, P. Yao, W. Tang, Q. Shi, and C. Zhang, "Complexity-guided container replacement synthesis," in *OOPSLA*, 2022. [Online]. Available: https://doi.org/10.1145/3527312

[12] T. McMichen, N. Greiner, P. Zhong, F. Sossai, A. Patel, and S. Campanoni, "Representing data collections in an ssa form," in *CGO*, 2024. [Online]. Available: https://doi.org/10.1109/CGO57630.2024.10444817

[13] G. L. Steele, *Common LISP: The Language*, 2nd ed., ser. HP Technologies. Saint Louis: Elsevier Science, 1984.

[14] C. Hanson and M. S. Team, *MIT/GNU Scheme Reference Manual*. London, GBR: Samurai Media Limited, 2015.

[15] (2022, Sep) Java development kit version 19 api specification. Accessed: May 14, 2025. [Online]. Available: https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/String.html

[16] lua-users wiki: Immutable objects. Accessed: May 14, 2025. [Online]. Available: http://lua-users.org/wiki/ImmutableObjects

[17] Clojure - special forms. Accessed: May 14, 2025. [Online]. Available: https://clojure.org/reference/special_forms

[18] Chapter 5. conversions and promotions. Accessed: May 14, 2025. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html

[19] (2024, aug) New unique package - the go programming language. Accessed: May 14, 2025. [Online]. Available: https://go.dev/blog/unique

[20] P. R. Calder and M. A. Linton, "Glyphs: flyweight objects for user interfaces," in *Proceedings of the 3rd Annual ACM SIGGRAPH Symposium on User Interface Software and Technology*, 1990. [Online]. Available: https://doi.org/10.1145/97924.97935

[21] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *CGO*, 2004. [Online]. Available: https://doi.org/10.1109/CGO.2004.1281665

[22] D. Lemire, G. Ssi-Yan-Kai, and O. Kaser, "Consistently faster and smaller compressed bitmaps with roaring," *Software: Practice and Experience*, vol. 46, no. 11, p. 1547–1569, Apr. 2016. [Online]. Available: https://doi.org/10.1002/spe.2402

[23] [Online]. Available: https://lucene.apache.org/index.html

[24] B. Chattopadhyay, P. Dutta, W. Liu, O. Tinn, A. Mccormick, A. Mokashi, P. Harvey, H. Gonzalez, D. Lomax, S. Mittal, R. Ebenstein, N. Mikhaylin, H.-c. Lee, X. Zhao, T. Xu, L. Perez, F. Shahmohammadi, T. Bui, N. McKay, S. Aya, V. Lychagina, and B. Elliott, "Procella: unifying serving and analytical data at youtube," *Proceedings of the VLDB Endowment*, 2019. [Online]. Available: https://doi.org/10.14778/3352063.3352121

[25] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson, "An experimental study of bitmap compression vs. inverted list compression," in *Proceedings of the 2017 ACM International Conference on Management of Data*, Chicago Illinois USA, 2017. [Online]. Available: https://doi.org/10.1145/3035918.3064007

[26] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *PLDI*, 2013. [Online]. Available: https://doi.org/10.1145/2491956.2462176

[27] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Tvm: an automated end-to-end optimizing compiler for deep learning," in *OSDI*, 2018. [Online]. Available: https://doi.org/10.5555/3291168.3291211

[28] M. P. Lücke, O. Zinenko, W. S. Moses, M. Steuwer, and A. Cohen, "The mlir transform dialect: Your compiler is more powerful than you think," in *CGO*, 2025. [Online]. Available: https://doi.org/10.1145/3696443.3708922

[29] A. Matni, E. A. Deiana, Y. Su, L. Gross, S. Ghosh, S. Apostolakis, Z. Xu, Z. Tan, I. Chaturvedi, B. Homerding, T. McMichen, D. I. August, and S. Campanoni, "Noelle offers empowering llvm extensions," in *CGO*, 2022. [Online]. Available: https://doi.org/10.1109/CGO53902.2022.9741276

[30] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *ISPASS*, 2009. [Online]. Available: http://iss.ices.utexas.edu/Publications/Papers/ispass2009.pdf

[31] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications," in *PACT*, 2008. [Online]. Available: https://doi.org/10.1145/1454115.1454128

[32] J. Leskovec and R. Sosič, "Snap: A general-purpose network analysis and graph-mining library," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, Jul. 2016. [Online]. Available: https://doi.org/10.1145/2898361

[33] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[34] Abseil. An open-source collection of core c++ library code. [Online]. Available: https://abseil.io/

[35] Y. Smaragdakis and D. Batory, "DiSTiL: A transformation library for data structures," in *Conference on Domain-Specific Languages (DSL 97)*, 1997. [Online]. Available: https://www.usenix.org/conference/dsl-97/distil-transformation-library-data-structures

[36] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv, "Data representation synthesis," in *PLDI*, 2011. [Online]. Available: https://doi.org/10.1145/1993316.1993504

[37] ——, "Concurrent data representation synthesis," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012. [Online]. Available: https://doi.org/10.1145/2254064.2254114

[38] M. A. Bender, A. Conway, M. Farach-Colton, W. Kuszmaul, and G. Tagliavini, "Tiny pointers," in *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2024. [Online]. Available: https://doi.org/10.1137/1.9781611977554.ch21

[39] N. Wanninger, T. McMichen, S. Campanoni, and P. Dinda, "Getting a handle on unmanaged memory," in *ASPLOS*, 2024. [Online]. Available: https://doi.org/10.1145/3620666.3651326

[40] C. Lattner and V. Adve, "Automatic pool allocation: improving performance by controlling data structure layout in the heap," in *PLDI*, 2005. [Online]. Available: https://doi.org/10.1145/1065010.1065027

[41] C. Lattner and V. S. Adve, "Transparent pointer compression for linked data structures," in *Proceedings of the 2005 Workshop on Memory System Performance*, 2005. [Online]. Available: https://doi.org/10.1145/1111583.1111587

[42] P. Zhao, S. Cui, Y. Gao, R. Silvera, and J. N. Amaral, "Forma: A framework for safe automatic

array reshaping," *ACM Transactions on Programming Languages and Systems*, 2007. [Online]. Available: https://doi.org/10.1145/1290520.1290522

[43] S. Curial, P. Zhao, J. N. Amaral, Y. Gao, S. Cui, R. Silvera, and R. Archambault, "Mpads: memory-pooling-assisted data splitting," in *Proceedings of the 7th international symposium on Memory management*, Tucson AZ USA, 2008. [Online]. Available: https://doi.org/10.1145/1375634.1375649

[44] J. Shirako and V. Sarkar, "An affine scheduling framework for integrating data layout and loop transformations," in *LCPC*, ser. Lecture Notes in Computer Science, 2022. [Online]. Available: https://doi.org/10.1007/978-3-030-95953-1_1

[45] T. McMichen and S. Campanoni, "Automatic Data Enumeration for Fast Collections," Nov. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.17872601