Representing Data Collections in an SSA Form

Tommy McMichen, Nathan Greiner, Peter Zhong, Federico Sossai, Atmn Patel, Simone Campanoni
Northwestern University
Evanston, IL, USA

***Abstract*—Compiler research and development has treated computation as the primary driver of performance improvements in C/C++ programs, leaving memory optimizations as a secondary consideration. Developers are currently handed the arduous task of describing both the *semantics* and *layout* of their data in memory, either manually or via libraries, *prematurely lowering* high-level data collections to a low-level view of memory for the compiler. Thus, the compiler can only glean conservative information about the memory in a program, e.g., alias analysis, and is further hampered by heavy memory optimizations. This paper proposes the Memory Object Intermediate Representation (MEMOIR), a language-agnostic SSA form for sequential and associative data collections, objects, and the fields contained therein. At the core of MEMOIR is a decoupling of the memory used to *store data* from that used to *logically organize data*. Through its SSA form, MEMOIR compilers can perform element-level analysis on data collections, enabling static analysis on the state of a collection or object at any given program point. To illustrate the power of this analysis, we perform dead element elimination, resulting in a 26.6% speedup on `mcf` from SPECINT 2017. With the degree of freedom to mutate memory layout, our MEMOIR compiler performs *field elision* and *dead field elimination*, reducing peak memory usage of `mcf` by 20.8%.**

***Keywords*—compilers, intermediate representation, optimization**

I. INTRODUCTION

Imperative programming languages require developers to describe their programs via direct updates to the program state. Some of these languages, namely C, give developers direct access to memory, making the ceiling for manual memory optimizations nearly unlimited. Using this degree of freedom, developers have been able to build operating systems, optimizing compilers, and interpreters.

However this manual control comes with the caveat that *all memory optimizations* must be created manually. This spawned mostly out of necessity, as compilers of the time were almost solely translation units, taking C as a portable assembly language and translating it to the target machine code. As such, developers were required to *prematurely optimize* [1] memory, before the compiler could perform meaningful optimizations.

For projects where performance is a primary goal, manual memory optimizations are prevalent throughout the source code. Anytime a developer wants to change a data structure, they must consider the implications of that change on existing memory optimizations. A daunting task, as memory optimizations are performed by careful consideration of both the data structure definition and its multitude of allocations. However this leaves compilers with lacking degrees of freedom, as these decisions are fixed *before compilation*.

As a result, production compiler optimizations either focus on scalar values or are limited in their applicability when memory is involved. Modern compilers seek to perform more aggressive transformations, such as automatic vectorization and parallelization [2–14], to fully utilize modern, multi-core processors. Such transformations require precise information about data and control dependencies in the program [15–17]. For programs operating on scalars, these dependencies can be easily analyzed with SSA forms [18,19]. However these techniques are severely limited when dealing with applications operating on complex data structures holding increasingly large amounts of data that must be stored in memory.

At present, only fixed-length arrays and objects have SSA forms [20,21]. Compilers, therefore, must rely on pointer analyses for data flow information about memory objects. This information can be improved by field-sensitive [22] and type-based [23] analyses, however common manual memory optimizations create spurious dependencies and ambiguity that the compiler cannot resolve. An example of this is allocation reuse, wherein a memory location is used to represent multiple objects over the execution of the program. This optimization is common for vectors, which may use the same memory location for different elements throughout its lifetime. This aggregates the disjoint lifetimes of individual elements into a single, long-lived lifetime. Through such premature optimizations, the compiler cannot distinguish between dependencies injected by the developer and those logically necessary.

The problems facing modern compilers are the culmination of ambiguous memory behavior and lacking degrees of freedom for dependency breaking transformations. To remedy this, the compiler requires *unambiguous memory operations* via strong guarantees about the type, allocation, and usage of memory within the program. Memory behavior must be presented in a form that can be meaningfully *analyzed* and *transformed*.

This paper proposes the Memory Object Intermediate Representation (MEMOIR). MEMOIR provides the compiler with an SSA representation for sequential and associative data collections. Additionally it defines a representation for objects and their fields. By decoupling the representation of **memory used to *store data* from the memory used to *logically organize data***, MEMOIR grants powerful guarantees for transformation and enables sparse data flow analysis for elements of collections and fields of objects via def-use chains. MEMOIR also grants the degrees of freedom necessary to change the memory layout of individual objects as well as the broader memory structure of a program. By providing an IR that is amenable to both analysis and transformation, MEMOIR compilers can emit performant code without placing the burden of memory optimization on developers.

This paper makes the following contributions:

- Makes the empirical observation that the majority of memory used within well-established C/C++ benchmark suites has a higher-level data structure (§III).
- Proposes MEMOIR, an SSA IR for associative and sequential collections, objects and their fields (§IV).
- Presents *dead element elimination*, *redundant indirection elimination*, and *field elision*, novel transformations that rely on element-level analyses (§V).
- Implements algorithms to generate and lower MEMOIR without incurring spurious copies (§VI).
- Evaluates the impact of MEMOIR compiler transformations on well-established C/C++ benchmarks (§VII).

Our MEMOIR compiler implementation is publicly available at https://github.com/arcana-lab/memoir.

II. BACKGROUND

Compilers rely on data flow analyses to perform optimizations, such as dead code elimination and common subexpression elimination. The SSA form [18] was introduced to provide a sparse representation for the data flow of variables throughout a program via def-use chains. This representation enables compilers to perform data flow analyses more efficiently and succinctly [24]. As such, SSA has become the *de facto* standard for modern, optimizing compiler intermediate representations (IR) such as LLVM [25] and GCC [26].

The Jalapeño compiler [27] sought to represent a subset of memory objects in Java programs with an SSA form. Their work made use of Array SSA form [20] to capture the semantics of fixed-size arrays and the Extended Array SSA form [21] to represent the fields of statically-, strongly-typed objects. This enabled the compiler to perform parallelization of array processing applications and load-store propagation for accesses to object fields by analyzing elements of these simple data collections at any given program point. While this work enabled compilers to more easily analyze simple array and object structures in a program, it does not provide a general representation for data structures that are sequential or associative. These data structures have operational semantics that cannot be represented as simple arrays or individual objects. Without abstracting these collections into a general form, the compiler is unable to provide a unified analysis.

In dealing with a similar problem, tensor compilers have provided an inspiring generalization for higher-dimensional array structures. Efficient memory layout of tensors is heavily tied to the structure of the data stored therein, commonly classified as either sparse or dense. Sparse tensors are commonly stored as tree structures to reduce memory usage, limiting the ability for compilers to perform optimizations because of the linked data structure. To remedy this, the TACO compiler [28–30] provides a general representation for tensor operations, abstracting away the storage structure as either sparse or dense. The compiler can thus perform unified analysis of tensors while still reasoning about the structure of the stored data when generating code, illustrating the power of generalized representations while still maintaining rich semantics.

III. MOTIVATIONS

At present, TACO and Jalapeño's representations do not generalize to common data structures. These include, linked lists, key-value stores, and memory allocations that grow and shrink throughout program execution. To illustrate the prevalence of these data structures in modern programs, we inspected the memory usage and accesses of C/C++ benchmarks in the SPECspeed 2017 Integer suite (SPECINT 2017).

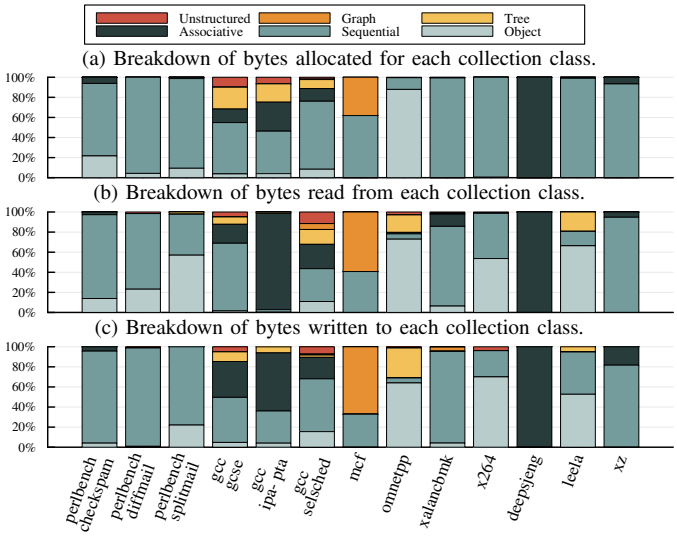

Fig. 1: Classification of SPECINT 2017 heap memory usage.

Figure 1 shows the breakdown of heap memory usage, as reported by Valgrind's dynamic heap analysis tool [31]. These benchmarks use a negligible amount of stack memory. We manually classify collections into one of six classes. *Sequential* collections are contiguous in index space, e.g., arrays, vectors and linked lists. *Associative* collections store a relation between key-value pairs and have an index space composed of the keys stored therein. *Objects* are fixed-length data structures composed of heterogeneously-typed fields. *Trees* and *graphs* represent any tree or graph-based data structure, making no distinction between their storage formats. *Unstructured* collections have no well-defined structure or are dictated by an externally defined memory layout, such as a file. Of note, the existing work on SSA forms described in §II does not fully cover sequences as Array SSA does not provide a representation for the reallocation of sequential collections; associative arrays, trees and graphs do not have any SSA representation; and unstructured data has no representation at present, aside from file readers represented as streams [32].

As illustrated above, modern applications utilize complex data structures. However, these data structures entail increasingly complex memory dependencies in the IR due to ambiguities in their representation. These memory dependencies result in overly-conservative analyses, limiting existing techniques and constricting the compiler's optimization space. This paper focuses on three sources of ambiguity in modern compiler representations. These are ***linked data structures***, ***stateful data accesses***, and ***premature memory optimizations***.

Linked data structures are the source of the undecideability of memory alias analysis [33]. These data structures hinder static analyses, as they are unable to determine with certainty where in the list a value came from. However, these data structures do possess an underlying index space that the compiler can use to disambiguate accesses. By representing linked lists as a sequence of values, similarly to an array, the index space can be exploited by the compiler. This comes from decoupling the representation of the memory used to *store the data* from the memory used to *logically organize the data*.

Stateful data accesses create data dependencies between unrelated memory operations. Listing 1 illustrates a piece of code that is *unoptimizable* by language-agnostic IR optimizations. Even though the key `0` is statically known to be unique from `1`, neither `clang++`, `g++`, nor `icpc` can propagate the constant `10` to the `return` statement: to the compiler every write to the map may update the state of the entire collection. In the code example, this is caused by a premature lowering of `std::unordered_map` to its implementation in the standard library.

Listing 1: Example of stateful data access in C++.

```
1 int work(std::unordered_map<int, int> &map) {
2   map[0] = 10;
3   map[1] = 11;
4   return map[0];
5 }
```

Memory optimizations performed by developers, such as over-provisioning of arrays, cause memory analyses to fail when details about element-level liveness is needed. This results from memory locations with disjoint lifetimes being aggregated into the same, long lived lifetime. As a result, the compiler cannot disambiguate accesses between unique objects, creating unresolvable false dependencies.

To remedy these shortcomings of the compiler we propose MEMOIR, an SSA representation for associative and sequential data collections, objects and the fields contained therein. MEMOIR provides a representation for the most commonly used memory classes in SPECINT 2017: objects, sequences and associative arrays. MEMOIR's SSA form enables the analyses needed by modern optimizations, which require a detailed understanding of the memory locations being accessed throughout the program. To guarantee this SSA form, we propose the use of the MEMOIR type system, which enforces static-, strong-typing for SSA collection variables.

IV. SSA Form for Collections and Objects

MEMOIR is an intermediate representation with **data collections as first-class citizens**. To enable element-level analysis and transformation on generalized data collections, MEMOIR provides three main properties for collections.

- Collections have a **static single assignment** (SSA).
- Collections are **value types**.
- Collections and their elements have **static, strong types**.

These properties make MEMOIR data collections immutable.

MEMOIR consists of named variables for collections, instructions for construction, access, and data flow of collections and objects, and a type system to represent object layout. MEMOIR instructions allow the creation of new collections from existing ones with unambiguous operations to *add*, *remove* or *redefine* elements. The syntax for MEMOIR collections, objects and types is shown in Figure 2.

A. Collections

We define a collection to be an *index-value mapping*. Indices and values stored within a collection have static, strong types. We refer to an individual index-value pair in the collection as an *element*. We refer to the set of indices present in the index-value mapping as the *index space* of a collection. In this paper we explore two variants of collections with different constraints on the index space, sequential and associative, explained in §IV-C and §IV-D, respectively. These variants harken back to the concepts of position- and value-dependent containers introduced by prior work [34].

B. Operations on SSA Collections

Reading elements of a collection is performed with the `READ` operator, of the form `v = READ(c, i)`, where `c` is the collection being accessed and `i` is the index being read from. The value held at that index is stored in the variable `v`. Reading from an uninitialized element is considered undefined behavior. Reading from an index not in the index space of the collection is similarly undefined behavior.

The *USEϕ*, as introduced in prior work [21], links accesses to the same collection in control flow order. This allows sparse data flow analyses to associate a lattice variable with *each access*, disaggregating use information from the definition. As *USEϕ*'s are not needed for every analysis, and increase program size by an additional instruction per-read, they can be constructed and destructed on demand via copy-folding [24].

Redefinition of elements in MEMOIR is accomplished by the *write* operation. Write operations are of the form c_1= `WRITE(`c_0`, i, v)`, where c_1 is a copy of the input collection c_0 with the exception of c_1`[i] = v`. With the write operation, a fixed-size collection is put in SSA form. Write operations in MEMOIR are similar to functional updates or *DEFϕ*'s, as introduced in prior work [20,21].

Insertion and Removal of elements in an SSA form is provided by the *insert* and *remove* operations. These operations are the *only* way changes to the index space of collections can be represented. *Insert* operations are of the form c_1= `INSERT (`c_0`, i, v)`, where c_1 is a copy of c_0 with index `i` inserted and c'`[i] = v` if `v` is defined. *Remove* operations, of the form c_1= `REMOVE(`c_0`, i)`, where c_1 is a copy of c_0 except for the element at index `i` being removed. The semantics of these operations depends on the type of collection being operated on, the details of which are expanded upon in §IV-C and §IV-D.

Copying Elements of a Collection is performed by the *copy* operation, of the form c_1= `COPY(`c_0`)`, which creates a new collection, c_1, with the same index-value mapping of c_0. Additional semantics for sequences are explained in §IV-C.

The Size of a Collection can be queried with the *size* operation, of the form `%n = size(`c_0`)`. The result of this query is the number of index-value pairs in the collection.

id ::= unique identifier, x ∈ Name

Types

```
    T ::=  PrimT | τid | &τid
PrimT ::=  i64 | i32 | i16 | i8
       |   u64 | u32 | u16 | u8 | bool
       |   index | f64 | f32 | ptr
CollT ::=  Seq<T> | Assoc<T,T>
 DefT ::=  type τid= { x: T, ...}
```

Variables

```
  idx ::=  i | elem
 elem ::=  %id: PrimT | obj
    i ::=  %id: index | end
  obj ::=  @id: RefT
    c ::=  seq | assoc | Fid: Assoc<&τid, T>
  seq ::=  Sid: Seq<T>
assoc ::=  Aid: Assoc<T,T>
```

Instructions inst ∈ Instructions

```
inst ::=  seq = new Seq<T>(i)
      |   assoc = new Assoc<T, T>
      |   obj = new τid | delete(obj)
      |   elem = READ(c, idx)
      |   c = USEϕ(c)
      |   c = WRITE(c, idx, elem)
      |   c = INSERT(c, idx, [elem]) | seq = INSERT(seq, i, seq)
      |   c = REMOVE(c, idx) | seq = REMOVE(seq, i, i)
      |   c = COPY(c) | seq = COPY(seq, i, i)
      |   seq[,seq] = SWAP(seq, i, [i,] [seq,] i)
      |   %id: index = size(c)
      |   %id: bool = HAS(assoc, elem)
      |   seq = keys(assoc)
      |   c = RETϕ(c,...) | c = ARGϕ(c,...) | c = ϕ(c,...)
```

Fig. 2: The syntax for MEMOIR types, collections, objects and instructions.

C. Sequences

A *sequence* is a collection of data organized with a contiguous index space. This index space is defined as the linear order of $\{n \in \mathbb{N} \mid 0 \leq n < l\}$, where l is the sequence's length. SSA sequence variables are denoted as `cid`, e.g., `S0: Seq<i32>` is a sequence with elements of 32-bit signed integers. More information about element types is included in §IV-E.

A new sequence with n elements of type T can be created via a `new Seq<T>(n)` instruction; `n` need not be statically known, but the length of the sequence is fixed upon allocation. Elements are uninitialized upon allocation. For operations on sequences, the `end` symbol is used as syntactic sugar for the `size` of the sequence being accessed.

On account of their sequential index space, sequences have additional semantics for the *remove*, *copy* and *swap* operations. A visual summary of these differences is shown in Figure 3 The first is the ability to operate on *ranges*, as the index space is contiguous. This is done by specifying a second index for the argument, which is the end of the range. We will use the shorthand `S[a:b]` to represent the range of a sequence from `a` to `b` (exclusive) when explaining the semantics of such operations. For example, `S1= REMOVE(S0,i,j)` creates `S1`, where `S1[0:i] = S0[0:i]` and `S1[i:end] = S0[j:end]`.

Additionally, the *insert* operation can insert elements from a sequence at a given index. For example, `S2= INSERT(S1, i, S0)`, which creates `S0`, where `S2[0:i]=S1[0:i]`, `S2[i:i+size(S0)]=S0[0:end]`, and `S2[i+size(S0):end]=S1[j:end]`.

The *swap* operation provides a means to swap ranges of sequences with one operation. For example, `S1= SWAP(S0,i,j,k)` creates `S1`, a copy of `S0` except for the ranges `i:j` and `k:k+j-i` swapped. Similarly, `S3,S2= SWAP(S1,i,j,S0,k)` swaps ranges `i:j` and `k:k+j-i` between sequences `S1` and `S0`.

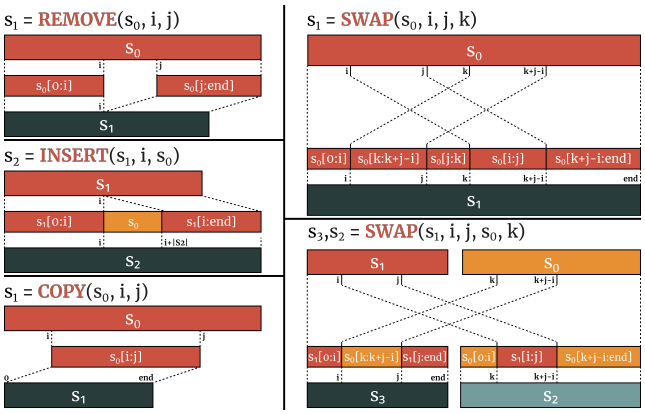

Fig. 3: Visualization of operations on sequence ranges.

D. Associative Arrays

An *associative array* is a mapping from keys to values. For example, `A0: Assoc<f32, bool>`, is an associative array of 32-bit floating point keys to boolean values.

In addition to the collection operations defined in §IV-B, the *has* operator, denoted `%h = HAS(A0, k)`, checks if the given key `k` is contained in the associative array `A0`. If it is, then a `true` value is written to the boolean variable `%h`, otherwise `false` is written. Additionally, the *keys* operator `S1= keys(A0)`, creates a sequence `S1` containing the keys of `A0`; there are no guarantees on the order of keys in `S1`.

Identity equality is used for key comparisons on primitive types. MEMOIR uses shallow equality for reference types, where keys alias iff they reference the same object. For object types MEMOIR checks equality for *each* element and field.

E. Field Arrays, Objects and Types

Objects flow throughout a program via *object references*, denoted as `@id`. Individual objects have explicit creation and deletion sites with the `new` and `delete` operators, respectively. Semantically, nested objects are stored as unique references within read-only elements of the collection, these references can be read into a variable via the `READ` operator.

Fields of objects are accessed via *field arrays*, borrowed from the concept of heap arrays [21]. Field arrays are an *associative array*, mapping an object reference to a field value. Each field of an object type–defined below–has a unique field array and are instantiated with the object type definition. By construction, a field array *cannot alias* with any other field of the object. This representation for fields allows MEMOIR

compilers to easily track all accesses of a given field, even when its owning object is an element of a collection. It also decouples the *access of fields* from the *layout of fields* within the object, enabling myriad degrees of freedom to MEMOIR compilers. Among these is the ability to reorder, add or delete fields within an object.

To specify the layout of objects and provide a mechanism for such transformations, MEMOIR includes the *object type*: an ordered list of individually addressable, typed data fields. For example, τ_T=`{a:i32, b:f32}` defines a new type τ_T containing two fields. The first having unique identifier `a` and is of 32-bit signed integer type. The second being `b` of 32-bit floating point type. The field arrays for them being $F_{T.a}$: `&`τ_T`→a` and $F_{T.b}$: `&`τ_T`→b`, respectively.

MEMOIR includes support for unsigned and signed integers (e.g., `i32`), booleans, floating point values, and C-style pointers. C-style pointers are included to support operations that require access to locations within conventional memory allocations, such as memory-mapped regions. Reference types are also supported, being nullable references to an object of a given object type. For example, `Seq<&`τ_T`>` is a sequence type containing elements that are references to structs of type τ_T.

Object types can contain nested object types, but may not be recursively defined. This ensures that object types have a finite, and statically-known, size in memory and a finite depth equality function when used as keys of associative arrays.

V. EMPOWERING THE COMPILER

By providing a more comprehensive representation for collections, MEMOIR enables automatic transformations on both the computation performed on data and its memory layout. In this section we will explore this optimization space.

We will operate on a constrained form of LLVM [25], where irreducible loops [35] are not permitted. The μ-operation [36] is used to represent a ϕ in the context of a loop; the first operand is the initial value and the second is used by later iterations. The **ARG**ϕ and **RET**ϕ functions are used for interprocedural data flow. *ARG*ϕs are attached to each parameter of a function, mapping a parameter to its incoming argument from each possible call site. $RET\phi$s are attached to each call function and map live-out variables from each possible return statement. During partial compilation, externally visible functions have an unknown operand from an unknown call site added to their *ARG*ϕs, similarly, $RET\phi$s of indirect calls have an unknown incoming value from an unknown function.

For analysis, we define an *expression tree*, representing an expression and the expressions that compose it, as well as a *range*, describing a contiguous subspace of a sequence *via* expression trees. We then define an analysis for the range of live elements in a sequence at each program point.

Def. 1. An *expression tree* is a tree where every internal node is an *operator* and every leaf is either a *variable* or a *constant*. We will define a partial ordering of expression trees, $t_1 \subseteq t_2$ iff t_2 contains t_1 as a subtree.

TABLE I: Constraint rules for sequences. Where $|S_i| \equiv$ **size**(S_i), and $\%a$, $\%b$ are new parameters of the function.

Operation	Constraints
$S_2 = \boldsymbol{\phi}(S_1, S_0)$	$S_2 \sqsubseteq S_1, S_2 \sqsubseteq S_0$
$... =$ **READ**(S_0, i)	$i + [0:1] \sqsubseteq S_0$
$S_1 =$ **USE**$\boldsymbol{\phi}(S_0)$	$S_1 \sqsubseteq S_0$
$S_1 =$ **WRITE**(S_0, i, v)	$S_1 \sqsubseteq S_0$
$S_1 =$ **INSERT**(S_0, i, v)	$S_1 \wedge [0:i] \sqsubseteq S_0$, $S_1 \wedge [i+1:end] - 1 \sqsubseteq S_0$
$S_2 =$ **INSERT**(S_1, i, S_0)	$S_2 \wedge [0:i] \sqsubseteq S_1$, $S_2 \wedge [i:i+\|S_0\|] - i \sqsubseteq S_0$, $S_2 \wedge [i+\|S_0\|:end] - \|S_0\| \sqsubseteq S_1$
$S_1 =$ **REMOVE**(S_0, i, j)	$S_1 \wedge [0:i] \sqsubseteq S_0$, $S_1 \wedge [j:end] \sqsubseteq S_0$
$S_1 =$ **COPY**(S_0, i, j)	$S_1 + i \sqsubseteq S_0$
$S_3, S_2 =$ **SWAP**(S_1, i, j, S_0, k)	$S_3 \wedge [0:i] \sqsubseteq S_1$, $S_3 \wedge [i:j] - i + k \sqsubseteq S_0$ $S_3 \wedge [j:end] \sqsubseteq S_1$ $S_2 \wedge [0:k] \sqsubseteq S_0$ $S_2 \wedge [k:k+j-i] - k + i \sqsubseteq S_1$ $S_2 \wedge [k+j-i:end] \sqsubseteq S_0$
$S_1 =$ **SWAP**(S_0, i, j, k)	$S_1 \wedge [0:i] + k \sqsubseteq S_0$ $S_1 \wedge [i:j] - i + k \sqsubseteq S_0$ $S_1 \wedge [j:k] \sqsubseteq S_0$ $S_1 \wedge [k:k+j-i] - k + i \sqsubseteq S_0$ $S_1 \wedge [k+j-i:end] \sqsubseteq S_0$
$S_2 =$ **ARG**$\boldsymbol{\phi}(S_1, S_0)$	$S_2 \sqsubseteq S_1$, $S_2 \sqsubseteq S_0$
$S_2 =$ **RET**$\boldsymbol{\phi}(S_1, S_0)$	$[\%a:\%b] = S_2$, $[\%a:\%b] \sqsubseteq S_1, [\%a:\%b] \sqsubseteq S_0$

Def. 2. A *range* of a sequence is a contiguous subspace of its index space $[\ell : u]$, where ℓ and u are expression trees.

Def. 3. A *range lattice* $\mathcal{L}$ is a lattice comprising n lattice points $\{\mathcal{L}_1, ..., \mathcal{L}_n\}$. Each lattice point $\mathcal{L}_i$ maps to a range $[\ell_i : u_i]$. These lattice points are partially ordered with the subset relation $\sqsubseteq$ with $\mathcal{L}_i \sqsubseteq \mathcal{L}_j$ iff $\ell_i \subseteq \ell_j$ and $u_i \subseteq u_j$.

Def. 4. The *disjunctive merge operator* $\vee$ unions the ranges of two lattice points, defined as:

$$\mathcal{L}_i \vee \mathcal{L}_j = [\ell_i : u_i] \vee [\ell_j : u_j] \doteq [min(\ell_i, \ell_j) : max(u_i, u_j)]$$

Def. 5. The *conjunctive merge operator* $\wedge$ intersects the ranges of two lattice points, defined as:

$$\mathcal{L}_i \wedge \mathcal{L}_j = [\ell_i : u_i] \wedge [\ell_j : u_j] \doteq [max(\ell_i, \ell_j) : min(u_i, u_j)]$$

Live Range Analysis computes ranges for the live elements of sequence variables in a program. This uses a similar approach as the range analysis described in [37], extending it to be context-sensitive and modifying it to operate on sequence variables. The analysis, detailed in Algorithm 1, operates on a *constraints graph* derived from a constraints system, the rules for which are defined in Table I, being a backwards propagation of liveness information.

Def. 6. A *constraints graph*, $G = (N, E, L, C)$, where: N is a set of vertices where n_a is a vertex; E is a set of edges where $\overrightarrow{n_a n_b}$ is an edge; $L : E \to S$ is a function from edges to constraints; $C : E \rightharpoonup c$ is a partial function from edges to call site c, representing *context-sensitive edges*.

Algorithm 1: Live range analysis.

in: S a set of constraints
in: V a set of *sequence* variables.
in: $R(i) = [\ell : u]$ a mapping from *index* var. i to *range*
▷ $R(i)$ result of an intraprocedural range analysis [38]
out: $\rho(v, c) = [\ell : u]$ a mapping from *sequence* variable v and call site c to a *range*

let $G = (N, E, L, C)$ ▷ A *constraints graph* from Def. 6
foreach $v \in V$ **do** create a vertex $n_v \in N$
foreach $i \in I$ **do** create a vertex $n_i \in N$, **let** $\rho(i) = R(i)$
foreach $s \in S$ s.t. $s = f(x) \sqsubseteq y$ **do**
  create an edge $\overrightarrow{n_x n_y} \in E$
  let $L(\overrightarrow{n_x n_y}) = f(x) \vee \rho(x) \sqsubseteq y$
foreach $s \in S$ s.t. $s = [\ell : u] \sqsubseteq x$ **do**
  create a vertex $n_{[\ell:u]} \in N$
  let $L(\overrightarrow{n_{[\ell:u]} n_x}) = [\ell : u] \sqsubseteq x$
foreach $s \in S$ s.t. $s = [\%a : \%b]$ **alias** x **do**
  create an edge $\overrightarrow{n_x n_{[\%a:\%b]}} \in E$
  let c be the call site **let** $C(\overrightarrow{n_x n_{[\%a:\%b]}}) = c$ **let** $L(\overrightarrow{n_x n_{[\%a:\%b]}}) = \rho([\%a : \%b], c) = \rho(x)$
call *resolve_cycle*(G)

function *resolve_cycle*(G)
  foreach $n_v \in G$ in topological order **do**
    if n_v is a trivial SCC [39] **then**
      $\rho(v) = \vee\{L(\overrightarrow{n_u n_v}) | \overrightarrow{n_u n_v} \in E\}$.
    else
      let SCC be the SCC containing n_v
      let $H = \{\overrightarrow{n_x n_y} \in SCC | C(\overrightarrow{n_x n_y})$ is undef.$\}$ ▷ H is the context-insensitive subgraph of SCC
      call *resolve_cycle*(n_w)
      foreach $n_v \in SCC$ s.t. $C(\overrightarrow{n_u n_v})$ is defined **do**
        let $\rho(v, C(\overrightarrow{n_u n_v})) = \rho(u)$
      foreach $n_v \in SCC$ s.t. $\rho(v)$ is undefined **do**
        let $\rho(v, \text{undefined}) = [0 : end]$

Dead Element Elimination Using the live slice range analysis, we describe the *dead element elimination* transformation in Algorithm 2. This transforms sequence access and construction to only operate on the live slice range, eliminating dead code. The *materialization function* (defined below) is used to perform available expression analysis [40] on an expression tree, constructing operations. Following dead element elimination, constant propagation and folding are applied, simplifying the if-else regions along with a conventional sink pass to move computation into its newly conditional execution.

Def. 7. The *materialization function* $M(e, p) \rightharpoonup v$ is a partial function, which analyzes an expression tree e at a given program point p, constructing the necessary operations and returning the resultant value v, which is either a variable or a constant. $M(e, p) = e$ iff e is a constant, a parameter of the function containing p, or a variable dominating p. $M(e, p) = g$ iff $\exists$ variable g which dominates p and has the same global value number [41] as e. $M(e, p) = op(M(e_1, p), ..., M(e_n, p))$ iff e is an expression tree such that $e_i \subseteq e$ for all $i = 1, ..., n$, op is an operation with no side effects and $M(e_i, p)$ are defined for all i. Otherwise, $M(e, p)$ is undefined.

Algorithm 2: Dead element elimination.

in: $\rho(v, c) = [\ell, u]$ ▷ Result of live range analysis (Algorithm 1)

for each $(v, c) = [\ell, u] \in \rho$ **do**
  if c is defined **then**
    let f be the function containing v
    create $f'(c)$, a copy of f for c, if it does not already exist
    let `v` = variable corresponding to v in f'
  if $M(\ell, v)$ **and** $M(u, v)$ are defined **then**
    create `%l` = $M(\ell, v)$, **create** `%u` = $M(u, v)$
    if `v = WRITE(S0, i, ...)` **then**
      create
        `if ((i >= %l) and (i < %u))`
          `v' = WRITE(S0, i, ...)`
        `Sp = φ(v', S0)`
      replace uses of `v` with `Sp`.
    if `v = INSERT(S0, i, ...)` **then**
      create
        `if (i < %u)`
          `S1 = INSERT(S0, i, ...)`
        `Sp = φ(S1, S0)`
      replace uses of `v` with `Sp`
    if `v = SWAP(S0, i, j, k)` **then**
      create
        `%from_live = i < %u and j <= %l`
        `%to_live = k < %u and k+j-i <= %l`
        `if (%from_live and %to_live)`
          `S1 = SWAP(S0, i, k)`
        `else if (%from_live)`
          `for %n = 0 to j - 1`
            `%kv = READ(S0, k + %n)`
            `S2 = WRITE(S0, i + %n, %kv)`
        `else if (%to_live)`
          `for %m = 0 to j - i`
            `%iv = READ(S0, i + %m)`
            `S3 = WRITE(S0, k + %m, %iv)`
        `% else // Do nothing.`
        `Sp = φ(S1, S3, S5, S0)`
      replace uses of `v` with `Sp`
    if `v = RETφ(...)` **then**
      Pass `%l` into `%a` and `%u` into `%b` at the call c

Listing 2: Abridged `mcf` implementation in MEMOIR.

```
1  type τ0 = { arc: rawptr, cost: i64 }
2  fn master()
3    Sorig = { Initialize }
4    do
5      Sold = μ(Sorig, Ssorted)
6      Snew = new Seq<τ0>(0)
7      do // Filter elements.
8        Snew0 = μ(Snew, Snew2)
9        %i = μ(0, %i+1)
10       @cur = Sold[%i]
11       if (call check_cost(@cur))
12         Snew1 = INSERT(Snew0, end, @cur)
13       Snew2 = φ(Snew1, Snew0)
14     while((%i+1 < size(Sold) and (%i+1 < B))
15     do // Append elements.
16       Snew3 = μ(Snew2, Snew4)
17       %j = μ(0, %j+1)
18       v = { Initialize }
19       Snew4 = INSERT(Snew3, end, v)
20     while(%j < K)
21     call qsort(Snew4, 0, size(Snew4)) ❶
22     Ssorted = RETφ(Sin, Sout)
23     @max = Ssorted[0]
24   while(call check_opt(@max))
```

Listing 3: Abridged quick sort implementation in MEMOIR.

```
1  fn qsort(S_in:Seq<τ0>, %lo:index, %hi:index)
2    S_in = ARGφ(S_10, S_p, S_ls)
3    if (%hi < %lo or (%hi-%lo) <= 1) return
4    %pv = S_in[%lo] // pivot value
5    while true // partition
6      %i = μ(%lo, %i')
7      %j = μ(%hi, %j')
8      S_1 = μ(S_0, S_2)
9      for %i' = %i + 1 to %j step 1
10       if S_1[%i'] > %pv break
11     for %j' = %j - 1 to %i' step -1
12       if S_1[%j'] < %pv break
13     if %i' >= %j' break
14     S_2 = SWAP(S_1, %i', %j')
15   S_p = SWAP(S_1, %lo, %j')
16   call qsort(S_p, %lo, %q)
17   S_ls = RETφ(S_in, S_out)
18   call qsort(S_ls, %q+1, %hi)
19   S_out = RETφ(S_in, S_out)
20   return
```

Listing 2 and Listing 3, a simplification of `mcf`'s hot code, are used for illustration. `check_cost` and `check_opt` are summarized computations with no side effects. For `mcf`, Listing 4 shows the optimized `qsort` function; not shown is the call at ❶ of `master` being transformed to pass `0` and `B`.

Listing 4: Optimized quick sort implementation in MEMOIR.

```
1  fn qsort(S_in:Seq<τ0>, %lo:index, %hi:index,
2           %a:index, %b:index)
3    { ... Lines 2-4 unchanged ... }
4    while { ... Lines 5-13 unchanged ... }
5      if %a <= %i' < %b and %a <= %j' < %b
6        S_dee0 = SWAP(S_1, %i', %j')
7      else if %a <= %i' < %b
8        %jv = READ(S_1, %j')
9        S_dee1 = WRITE(S_1, %i', %jv)
10     else if %a <= %j' < %b
11       %iv = READ(S_1, %i')
12       S_dee2 = WRITE(S_1, %j', %iv)
13     S_2 = φ(S_dee0, S_dee1, S_dee2, S_1)
14   if %a <= %lo < %b and %a <= %j' < %b
15     S_dee3 = SWAP(S_1, %lo, %j')
16   else if %a <= %lo < %b
17     %pv = READ(S_1, %j')
18     S_dee4 = WRITE(S_1, %lo, %pv)
19   else if %a <= %j' < %b
20     %lov = READ(S_1, %lo)
21     S_dee5 = WRITE(S_1, %q, %lov)
22   S_p = φ(S_dee3, S_dee4, S_dee5, S_3)
23   { ... %a, %b passed into qsort calls ... }
```

Field Elision is a novel optimization enabled by the element-level analysis and decoupled field semantics of MEMOIR. Field elision converts a field of an object into a key-value pair stored in an associative array. This reduces the memory usage of possibly unused fields and increases the spatial locality of definitely used fields. Data structure splicing [42] pursues a similar goal by migrating fields that are not accessed with their co-located fields frequently to their own object. However, this entails additional pointer fields to locate the migrated fields, increasing the size of objects. Field elision avoids this by introducing a collection instead of pointer fields.

If a field is deemed a candidate for elision via affinity analysis [43,44], the transformation is applied. To apply this for a given candidate `T.a`, with field array $F_{T.a}$: `&T→U`; construct $A_{T.a}$ `= new Assoc<T,U>` at the beginning of the program's entry function. For each reference to $F_{T.a}$ replace it with $A_{T.a}$. If $F_{T.a}$ is used by an ARGϕ, create a new parameter in the corresponding function, passing $A_{T.a}$ as an argument, replacing all uses of $F_{T.a}$ in the function with the new parameter. Finally, remove field `a` from the definition of `T`.

Redundant Indirection Elimination (RIE) simplifies indirect accesses (e.g., `a[b[i]]`) [45] to associative arrays when the *index is derived from constant data* (e.g., elements of `b` are constant). When detected, the keys of an associative array can be replaced with the indices of a sequence or the keys of another associative array. This transforms the collection's type, removing an access to the index collection (e.g., `b[i]`).

The analysis is as follows: for an associative array, A_0`= new Assoc<T,U>`, let R be the set of variables in the def-use chain of A_0, let R be the set of these variables. If any variable in R *may*, but not *must*, reference A_0 (e.g., control divergence), RIE is not applicable. Otherwise, the transformation checks all accesses `r[k]`[1], where `r` $\in R$. If all keys `k` are of the form `k = READ(c, i)`, where `c` is a sequence or associative array and all instances of `c` *must* reference the same collection, then:

1) If `c` is a sequence, construct `c' = new Seq<U>(size(c))`.
2) If `c` is an assoc. array, construct `c' = new Assoc<V,U>`, where `V` is the key type of `c`.
3) $\forall$`r` $\in R$, replace accesses `r[k]` with `c'[i]`, where `k=c[i]`.

Dead Field Elimination utilizes a simple data flow analysis provided by MEMOIR's element-level analysis to eliminate fields of objects that are trivially dead. If a field array is never read from, i.e., it is never redefined with a *USE*ϕ and is never passed into an unknown function during partial compilation, it is deemed dead. If so, all writes to the field array and all variables in its def-use chain are removed and the field is eliminated from the type definition.

VI. A MEMOIR COMPILER

We will now describe our implementation of a MEMOIR compiler, the pipeline of which is laid out in Figure 4. First, we introduce the MUT library, a C/C++ programming interface for developers to grant the compiler with guarantees and degrees of freedom necessary to construct and optimize a MEMOIR program. Second, we present the SSA construction algorithm, converting the mutable collections in MUT into immutable SSA collections in MEMOIR. Finally, we discuss the lowering algorithms employed to generate LLVM IR, namely SSA destruction and heap/stack selection.

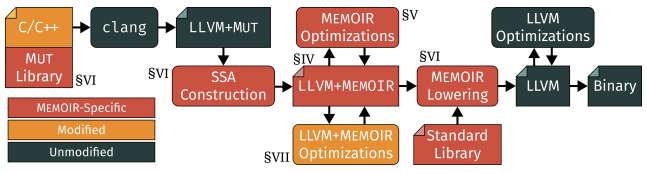

Fig. 4: Prototype MEMOIR compilation pipeline.

[1]The notation `c[i]` refers to any read, write or has operation that uses collection `c` at index `i`.

MUT	MEMOIR
`%n = size(c)`	⇒ `%n = size(c)`
`%v = read(c, i)`	⇒ `%v = READ(c, i)`
	`[ c' = USEφ(c) ]`
`write(c, i, v)`	⇒ `c' = WRITE(c, i, v)`
`insert(c, i)`	⇒ `c' = INSERT(c, i)`
`remove(c, i)`	⇒ `c' = REMOVE(c, i)`
`c2 = copy(c)`	⇒ `c2 = COPY(c)`
	`[ c' = USEφ(c) ]`
`s = new Seq<T>(n)`	⇒ `s = new Seq<T>(n)`
`remove(s, i, j)`	⇒ `s' = REMOVE(s, i, j)`
`insert(s, i, s2)`	⇒ `s' = INSERT(s, i, s2)`
	`[ s2' = USEφ(s2) ]`
`append(s, s2)`	⇒ `s' = INSERT(s, end, s2)`
	`[ s2' = USEφ(s2) ]`
`swap(s, i, j, k)`	⇒ `s' = SWAP(s, i, j, k)`
`swap(s, i, j, s2, i2)`	⇒ `s',s2' = SWAP(s, i, j, s2, k)`
`s2 = split(s, i, j)`	⇒ `s2 = COPY(s, i, j)`
	`s' = REMOVE(s, i, j)`
`s2 = copy(s, i, j)`	⇒ `s2 = COPY(s, i, j)`
	`[ s' = USEφ(s) ]`
`a = new Assoc<K,V>`	⇒ `a = new Assoc<K,V>`
`%b = contains(a, k)`	⇒ `%b = HAS(a, k)`
	`[ a' = USEφ(a) ]`
`s = keys(a)`	⇒ `s = keys(a)`

Fig. 5: MUT operations (left) and their mapping to MEMOIR operations (right). *USEφ*'s are only constructed on demand.

The MUT Library. To instantiate a MEMOIR program, the compiler relies on guarantees for allocation, access and typing of data collections, objects and elements. Similar to past language extensions [46,47], we use a library-compiler codesign to achieve this. To this end, we present the MUT library, consisting of sequences, associative arrays and objects. The MUT library contains explicit operators to directly mutate these collections—outlined in Figure 5—mirroring those available in the standard C++ library. Collections and objects in MUT have the same type properties as those laid out for MEMOIR in §IV, namely that they have strong, static types and mutable value semantics [48]. By employing the MUT library in their program, developers provide the guarantees and degrees of freedom necessary to construct and optimize a MEMOIR program. Additionally, by providing an extension rather than a replacement for C/C++, existing programs can be incrementally ported to benefit from MEMOIR.

SSA Construction. We now present the algorithm for constructing the SSA form of MEMOIR. We will only cover construction of SSA collections; construction of field arrays for object accesses is performed in the same manner as prior work [21]. For collections, a conventional ϕ insertion using the dominance frontier is used to insert ϕ-functions for MUT operations. A depth-first traversal of the CFG dominator tree is performed, applying the rewrite rules in Figure 5 are to MUT operations. Reaching definitions are updated accordingly: $ReachDef(v') = ReachDef(v)$ and $ReachDef(v) = v'$ for each v, v' pair in the rewrite rule.

Algorithm 3: Algorithm to destruct the SSA form.

```
in: f a MEMOIR function
for B ∈ preorder DFS of f's CFG dominator tree do
  foreach I = Def(v) in storage order of B do
    if v = new Seq(n) then UPDATE(v, v[0:n])
    else if v = keys(A) then UPDATE(v, v[0:end])
    else if v = φ(S_0, ..., S_n) then
      construct S_v = φ(S'_0, ..., S'_n),
        f_v=φ(f_0, ..., f_n), and
        t_v= φ(t_0, ..., t_n), where S_i= S'_i[f_i, t_i]
      UPDATE(v, S_v[f_v:t_v])
    else if v = USEφ(c_0) then replace uses of v with c_0
    else ▷ Handle cases where operation mutates collection.
      if S_i ∈ Operands(I) is not dead after this use then
        replace use of S_i with COPY(S_i)
      if v = WRITE(c_0, i, ...) then
        replace uses of v with c_0
      if v = REMOVE(c, ...) then
        construct remove(c, ...)
        replace uses of v with c
      if v = COPY(S, a, b) then UPDATE(v, S[a:b])
      if v = INSERT(S_0, i, S_1) then
        construct insert(S_0, i, S_1)
        replace uses of v with S_0
      if v = SWAP(S_0, i, j, k) then
        construct swap(S_0, i, j, k)
        replace uses of v with S_0
      if v,w = SWAP(S_0, i, j, S_1, k) then
        construct swap(S_0, i, j, S_1, k)
        replace uses of v with S_0 and uses of w with S_1
  function UPDATE(S, S'[a:b])
    foreach use U of S do
      if U=S[c:d] then replace S[c:d] with S'[a+c:a+d]
      else replace S with S'[a:b]
  function COPY(c: Collection)
    if c =S_0[i:j] then return copy(S_0, i, j)
    else return copy(c)
```

SSA Destruction. After SSA construction and optimizations are applied, MEMOIR collections are lowered to LLVM IR. SSA destruction, shown in Algorithm 3, coalesces collection variables, replacing SSA operations with ones that operate directly on their memory representation. Just as in scalar SSA, great care must be taken in destructing MEMOIR, as a naïve approach could drastically increase the number of allocations and copies in the program. Our MEMOIR compiler employs an SSA destruction algorithm with a focus on avoiding such *spurious copies*. These operations act directly on collections and *views* of sequences, which represent a contiguous subset of their index space, denoted `S[f:t]`. A variable being dead or alive refers to the program point following the instruction.

Collection Lowering. Finally, **new** operators are lowered to either a heap or stack allocation using the implementation in the standard library, e.g., `std::vector` or `std::map`. If an escape analysis computed on a **new** operator indicates that the collection or object is dead at all exit points of its containing function, it will be allocated on the stack; otherwise it is allocated on the heap.

VII. Evaluation

To evaluate MemOIR's construction, destruction and optimizations we have implemented a MemOIR compiler on the production-quality LLVM 9.0.0 compiler infrastructure, allowing us to utilize the open-source NOELLE [16,49] compiler framework for its loop-level analysis. The developer effort of our compiler implementation is shown in Table II. We have also manually converted the hot collections, which spread throughout much of the codebase, from SPECINT 2017's mcf_s, deepsjeng_s and LLVM's `opt` to use collections and objects from the Mut library. We evaluate the compilation time and effectiveness of our MemOIR compiler (§VII-B) and the effectiveness of MemOIR optimizations (§VII-C). We also analyze LLVM passes that MemOIR could improve (§VII-D).

A. Experimental Setup

Our evaluation was performed on a server with two Intel Xeon Gold 6258R (Cascade) processors running at 2.70GHz. Each processor has 28 cores with 2-way hyper-threading, 32KiB 8-way L1d$ and 1MiB 16-way L2$, backed by 38.5MiB 11-way LL$, all having a 64B line size. They are supported by 512GiB of DDR4 main memory running at 2933MT/s. The OS used is Red Hat Enterprise Linux 8.7.

We evaluate against four production-quality compilers: GCC 8.5.0 (**GCC**), ICC 18.0.1 (**ICC**), LLVM 9.0.0 (**LLVM9**), and LLVM 14.0.6 (**LLVM14**). All performance results are gathered from 10 executions per configuration, all values are the median of those executions relative to **LLVM9**.

B. Compilation

We evaluated the effectiveness of our compiler at avoiding spurious copies and providing compilation time on par with modern optimizations. Table III shows the breakdown of our MemOIR compiler's performance on mcf and deepsjeng from SPECINT 2017 as well as LLVM's `opt` middle-end compiler. Compilation time for solely perfoming SSA construction and destruction (MemOIR `O0`) is on par with that of `clang -O0`. Compiling with all MemOIR optimizations *and* `opt -O3` causes a reasonable increase in compilation time and could be improved with further engineering. We also show that *no spurious copies are introduced*.

TABLE II: MemOIR passes require reasonable developer effort.

MemOIR	*SLOC*	LLVM	*SLOC*
DEE	1211	NewGVN	2814
DFE	267	Sink	181
FE	580	ConstantFold	1788
RIE	461		

TABLE III: Our MemOIR compiler has reasonable compilation time with no spurious copies added by SSA construction.

	Compile Time (ms)				*# Collections*		
	MemOIR		LLVM				
Benchmark	O0	O3	O0	O3	Source	SSA	Binary
mcf	70.6	776.4	20.9	663.2	5	13	5
deepsjeng	246.0	1867.6	34.8	852.8	2	14	2
LLVM opt	225.9	668.4	52.0	414.7	8	37	8

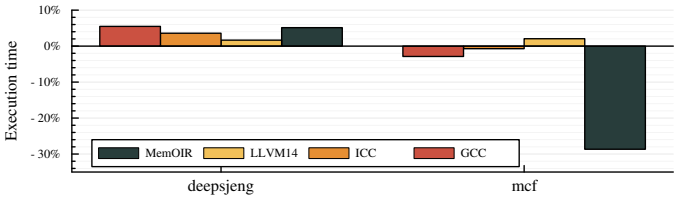

Fig. 6: Relative execution time of ported SPECINT 2017 benchmarks

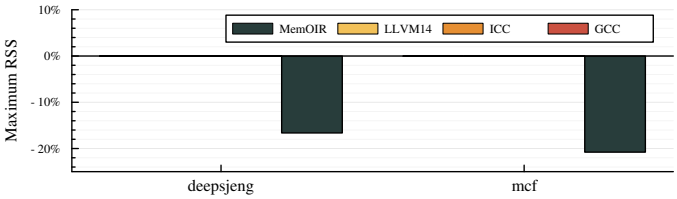

Fig. 7: Relative memory usage of ported SPECINT 2017 benchmarks.

C. Performance and Memory Usage Impact of Optimizations

We evaluated the impact of MemOIR optimizations on mcf_s and deepsjeng_s from SPECINT 2017. To evaluate the impact of the optimizations described in §V we utilize a separate compilation pipeline. The aforementioned programs are manually transformed, following the algorithms described in §V. These programs are then compiled using `clang 9.0.0`, with optimization level `O3`. By transforming these optimizations manually instead of automatically, we are able to isolate their impact without additional noise caused by different collection implementations from the baseline, which may stress different regions of the program.

We evaluate multiple permutations of MemOIR compiler transformations: Dead Element Elimination (**DEE**), Dead Field Elimination (**DFE**), Field Elision (**FE**), and Redundant Indirection Elimination (**RIE**). The **ALL** configuration has all of the above transformations applied. Evaluation of LLVM's `opt` is not included as the MemOIR optimizations explored in this paper were not applicable. Figures 6 and 7 show the execution time and maximum resident set size (max RSS), respectively, with the **ALL** configuration. We see that MemOIR optimizations are able to reduce the memory usage of both mcf and deepsjeng, while only mcf sees a reduction in execution time. For deepsjeng, only field elision and key folding were applicable, eliding a 16-bit field from the hottest data structure. This allowed for better packing of the struct in memory, reducing the memory usage by 16.6% but entailing a 5.1% increase in execution time due to cache performance.

All MemOIR optimizations were applicable to mcf, we will now explore their individual impact and interplay. **DEE** results in a 26.6% speedup over LLVM9 by reducing the computational complexity of mcf's quick sort on a sequence of length $n = K + B$ from $O(nlog(n))$ to $O(n + Blog(B))$. **FE**, in the absence of other optimizations, causes a 10.4% increase in execution time and a 3.3% increase in max RSS, where a single pointer field is elided. The resulting associative array is lowered to a hashtable, whose key-value store causes increases memory usage and expansion of the table causes the increased execution time. When combined with **RIE**, the positive impact

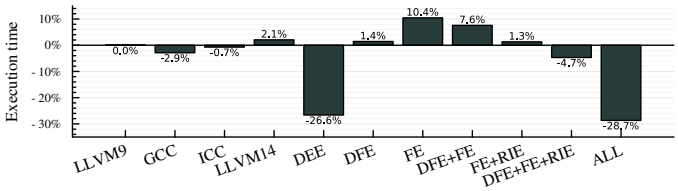

Fig. 8: Relative execution time for breakdown of mcf optimizations.

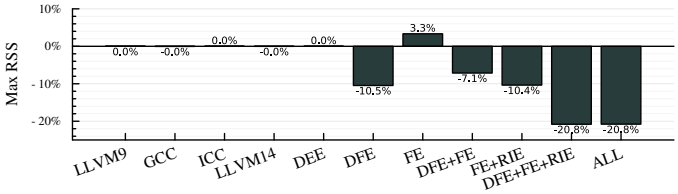

Fig. 9: Relative memory usage for breakdown of mcf optimizations.

of **FE** can be seen with a 10.4% decrease in max RSS with only a 1.3% increase in execution time, as the associative array is converted into a single sequence. This removes the storage of the key, only needing to store the value.

When combined with **DFE** the memory size of the object is shrunk to 56 bytes, allowing fields of more than one object to be stored on the same cache line. Because accesses to these objects are filtered by the value of the first field, filtered iterations (i.e., where the rest of the object is not read), can be processed along with the previous iteration, achieving a 4.7% speedup over the baseline. This combination also greatly reduces the memory usage of the benchmark, reducing the max RSS by 20.8%, approximately 844MiB. When **ALL** optimizations are applied an additional 2.1% speedup is seen over solely **DEE** and max RSS reduction remains 20.8%.

D. Pass Analysis

We evaluated optimizations in the LLVM 16.0.3[2] compiler pipeline to locate areas that traditional compiler techniques could be improved by MEMOIR, the goal being to find an upper-bound for such benefits. Results were gathered via manually inserted counters, from an invocation of `opt -O3` on the whole-program bitcode generated by `gclang`.

Global Value Numbering is restricted by LLVM's opaque memory locations, which prevent assignment of memory and pointer operations to existing congruence classes, introducing a large number of memory-related value numbers, as seen in Figure 10. With element-level information, these memory locations can be mapped to a smaller set of congruence classes.

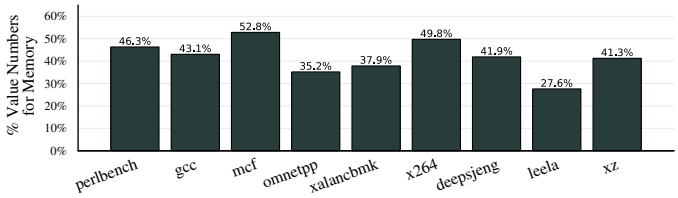

Fig. 10: Percentage of global value numbers introduced for memory operations in LLVM's `NewGVN` pass.

[2]LLVM 16.0.3 was the most recent release at the time of experimentation.

Sink is constrained by memory operations, which create barriers that instructions can not be safely moved across. Figure 11 illustrates how common of an occurence this is, as many attempts to sink an instruction fail because of instructions that may write or reference the same memory location. With unambiguous representations for *what* memory operations are being performed and *which* elements are being operated on, MEMOIR could enable additional, safe, code motion.

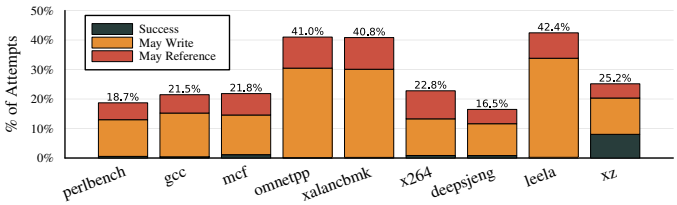

Fig. 11: Analysis of LLVM's `Sink` pass.

Constant Folding is blocked by an inability to propagate constants across memory barriers, the breakdown of which is seen in Figure 12. The element-level analysis provided by MEMOIR allows constants to be propagated along a collection's DEF-USE chain, an optimization performed in [21] for fields. Prior work has also proposed conditional constant propagation algorithms for Array SSA [50], which could be repurposed by MEMOIR compilers.

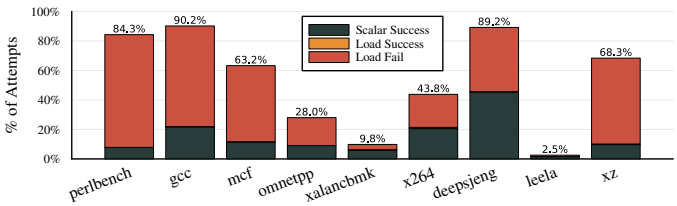

Fig. 12: Analysis of LLVM's `ConstantFold` pass.

VIII. Related Work

A. Compiler Intermediate Representations

Array SSA [20] provides a first class representation for arrays, but fails to generalize to sequential collections that grow and shrink throughout execution. Extended Array SSA [21] provides a representation for accessing fields of objects with static, strong types, introducing what this paper calls *field arrays*. MEMOIR generalizes both of these representations, making it amenable to techniques that use them [27,50–53].

Memory SSA [54] and similar works [55,56] provide a sparse representation of *points-to information*, enabling flow analyses on memory blocks. However, they do not grant degrees of freedom to the compiler as they do not provide a representation for the *structure of data* in memory.

MLIR [57] provides a compiler infrastructure for progressive lowering from higher- to lower-level representations. MLIR's `memref` dialect represents memory objects, but is purely a wrapper for the LLVM memory representation, providing no additional guarantees as the underlying memory is mutable. MLIR seems a useful substrate for MEMOIR compilers, but the concepts introduced by MEMOIR are missing, making its introduction a matter of engineering. Additionally, there is

currently no MLIR-based compiler capable of meaningfully handling the complexity of SPEC benchmarks.

SDFG's [58] represent the program as a stateless, acyclic graph of data containers and computation, wrapped by stateful nodes. This representation does not provide meaningful abstractions for their containers, suffering from the memory ambiguities shown in §III, especially stateful data accesses which require each access be wrapped in a stateful node. Similarly, HPVM [59] provides a dataflow graph representation of the program, with explicit data movement between nodes. These nodes may have side effects, which are modeled with the same low level memory representation as LLVM, making HPVM subject to the same memory aliasing problems.

B. SSA Destruction and Register Allocation Techniques

SSA destruction of data collections in MEMOIR bears a strong resemblance to the register allocation problem, known to be NP-complete [60]. We present an algorithm with a primary focus on avoiding spurious copies, simply preserving the user-defined allocation. Adapting strategies from scalar variable allocation [61–64], could enable MEMOIR compilers to optimize the allocations. Advanced escape analyses could be used to improve the stack allocation algorithm [65–67].

C. Precise Reasoning with Collections and Structured Heaps

There is a growing body of work related to the static analysis of data collections. Dillig et al. [34] proposed a framework for analyzing the contents of containers–analogous to element-level analysis in this paper–for the purpose of verification. Unifying their framework with MEMOIR provides an exciting opportunity for sound collection lowering to implementations outside of the standard library. More recently, structured heaps [68] have been proposed as a means of enabling static analysis for programs using dynamic memory, to which MEMOIR bears a strong, albeit higher-level, resemblance.

D. Programming Languages Amenable to MEMOIR

At its core, MEMOIR proposes collections as value types. In this paper, we implement a library in C/C++ to provide this functionality, however many languages exist which provide the guarantees needed for a MEMOIR compiler. Languages with mutable value semantics [48], which degrades references to second-class citizens, are amenable to SSA construction, as they are analogous to our MUT library. Such languages include Swift's `struct` types [69] and Hylo [70].

Languages with single-ownership, i.e., "borrowing", which guarantee that only one mutable reference will exist at a time can be used to construct a MEMOIR program. An example of this is Rust [71], which is steadily entering the programming *zeitgeist*. Similarly to Rust, newer languages such as Mojo [72] and Vale [73] have similar ownership models. Of note, *use* ϕ's cannot be constructed for these languages, as multiple *immutable* references may exist at once. While the aforementioned languages are promising directions of future work, the lack of accepted benchmark suites implemented in them, unlike C/C++, was deemed too large a barrier to adoption in our research at present.

Collection-oriented languages [74–77] have existed for many years now. APL [74] and SETL [77] serve as prime examples of their philosophy, focusing on arrays and sets, respectively, as prime concepts of the language. As such they are interesting source languages for compilation and, furthermore, their implementations provide a wealth of resources on optimizing collection-oriented programs [78]. A recent example of these concepts being exploited outside of their original languages is parallel block-delayed sequences [79], which implements loop-fusion techniques on sequences as a library for Parallel ML and C++. Investigating the extent these optimizations could be performed statically with MEMOIR provides an interesting starting point for this line of research.

IX. CONCLUSION

This paper introduced MEMOIR, a compiler intermediate representation in an SSA form for data collections and objects stored in memory. The core of MEMOIR comes from a decoupling of the memory used to *store* data from the memory used to *logically organize* data. Through this decoupling, the compiler is granted a generalized representation of both sequential and associative collections with well-defined operational semantics that cover common operations performed on them. This paper also introduced a prototype MEMOIR compiler, which is capable of performing element-level analysis on collections thanks to its SSA form and unambiguous operations. Using this analysis, the compiler is able to perform novel memory optimizations that must be applied manually today, including efficient layout of fields, selection of heap or stack allocation and copy elision. Furthermore, MEMOIR enables traditional, scalar analyses and transformations be applied to elements of collections. As an example, we generalized live variable analysis to be live range analysis and used it to perform dead element elimination, a generalization of dead variable elimination. With additional work on MEMOIR compilers, we believe that developers can be liberated from the burden of performing low-level, manual memory optimizations. Additionally, the avoidance of premature lowering and optimization of data collections can remove barriers to optimizations, improving the applicability of traditional compiler optimizations.

ACKNOWLEDGEMENTS

We thank members of the ARCANA Lab for their support and feedback on this work. We also thank the anonymous reviewers for insightful comments and invaluable suggestions, which made this work stronger. This material is based upon work supported by the U.S. Department of Energy under the contract number DE-SC0022268. This material is also based upon work supported by the National Science Foundation under Grants NSF-2119069, NSF-2107042, NSF-2028851, NSF-1908488. 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 prototype MEMOIR compiler described and evaluated in the paper. In addition to this it includes source-code patches for SPEC CPU2017 and LLVM opt. When applied, these patches port the programs to utilize MEMOIR collections and apply MEMOIR optimizations at the source-code level.

B. Artifact Check-list (meta-information)

- **Algorithm: SSA Construction, SSA Destruction, Dead Element Elimination, Dead Field Elimination, Field Elision, Redundant Indirection Elimination**
- **Program: LLVM; Optional: SPEC2017**
- **Compilation: LLVM9; Optional: LLVM14, GCC, ICC**
- **Transformations: MEMOIR optimizations and porting implemented as `ed` scripts.**
- **Binary: None**
- **Data set: SPEC CPU2017 Integer (not included)**
- **Run-time environment: Linux**
- **Hardware: Tested on Intel and AMD x64 machines.**
- **Metrics: Execution time, Max resident set size, Compilation time, Number of collections, Significant lines of code**
- **Output: Tables 2 and 3, Figures 5, 6, 7 and 8.**
- **Experiments: Performance and Compiler Evaluation**
- **How much disk space required (approximately)?: Generated artifact is 11GiB**
- **How much time is needed to prepare workflow (approximately)?: 30 minutes**
- **How much time is needed to complete experiments (approximately)?: 45 minutes (1 run, with minimal configurations enabled), 4 hours (1 run, with all configurations enabled)**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: MIT License**
- **Workflow framework used?: Unix Makefiles, Bash, customization described below.**
- **Archived?: Yes, https://doi.org/10.5281/zenodo.10182391**

C. Description

1) How delivered: The artifact is available on Zenodo, at https://zenodo.org/records/10201049.

2) Hardware dependencies: An x64 processor. This artifact has been tested on both Intel and AMD x64 machines.

3) Software dependencies: Running the artifact requires LLVM 9.0.0, Julia 1.9.4, scc, and gclang. It optionally depends on LLVM 14.0.6, GCC 8.5.0, and ICC 18.0.1.

All of the above dependencies are handled when building the docker image. This artifact has been tested with both docker 24.0.5 and Podman 4.6.1.

D. Installation

We have provided a Dockerfile to handle dependencies. If evaluating the artifact with SPEC2017, the easiest solution is to copy your `SPEC2017.tar.gz` into the artifact directory, which will be transfered to your docker container. If you choose to use docker, run the following two commands from within the installed artifact directory:

```
docker build -t cgo24-artifact .
docker run -it cgo24-artifact
```

After installation, run `make config` within the artifact directory to configure the artifact. Details about the configuration are included in §G.

E. Experiment workflow

After installation and configuration, run `make all`, which will run the following pipeline:

1) `make memoir-setup` compiles MEMOIR.
2) `make benchmark-setup` sets up benchmark files.
3) `make benchmark` compiles benchmarks.
4) `make performance` executes performance tests.
5) `make figures` gathers statistics about the MEMOIR compiler and then creates figures.

F. Evaluation and expected result

As the output of the experiment flow, the `figures/` directory will be populated with the tables and figures from the paper.

Table 2 (`figures/table_2.txt`) evaluates our prototype MEMOIR compiler. The first portion evaluates the compilation time for the MEMOIR compiler and LLVM compiler using no optimizations (O0) and all optimizations (O3).

Table 3 (`figures/table_3.txt`) evaluates the development effort of our prototype MEMOIR compiler. The SLOC for LLVM passes is not included.

Figure 5 (`figures/figure_5.pdf`) evaluates the impact of MEMOIR optimizations on execution time.

Figure 6 (`figures/figure_6.pdf`) evaluates the impact of MEMOIR optimizations on memory usage (max RSS).

Figure 7 (`figures/figure_7.pdf`) evaluates the impact of each MEMOIR optimization, in isolation and concert, on execution time.

Figure 8 (`figures/figure_8.pdf`) evaluates the impact of each MEMOIR optimization, in isolation and concert, on memory usage (max RSS).

G. Experiment customization

The artifact can be customized by running `make config`. The configuration options are as follows:

- **SPEC2017** can be enabled/disabled. If disabled, figures 5, 6, 7 and 8 cannot be generated.
- **Sweeping Compilers** can be enabled/disabled. When enabled, figures 5, 6, 7 and 8 will include performance evaluation for GCC and LLVM14.
- **Sweeping Optimizations** can be enabled/disabled. When enabled, the figures 7 and 8 will be generated.
- **Number of Runs** can be set, this will run each configuration of each benchmark that many times.

More detailed information about the artifact and how to extend it to be used on *new benchmarks and compiler configurations* is available in the artifact `README.md`.

H. 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] D. E. Knuth, "Structured programming with go to statements," *ACM Comput. Surv.*, vol. 6, no. 4, p. 261–301, dec 1974. [Online]. Available: https://doi.org/10.1145/356635.356640

[2] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks, "HELIX: Automatic parallelization of irregular programs for chip multiprocessing," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12, 2012, p. 84–93. [Online]. Available: https://doi.org/10.1145/2259016.2259028

[3] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, "Automatic thread extraction with decoupled software pipelining," in *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*, 2005.

[4] S. Campanoni, T. M. Jones, G. Holloway, G. Y. Wei, and D. Brooks, "HELIX: Making the extraction of thread-level parallelism mainstream," *IEEE Micro*, vol. 32, no. 4, pp. 8–18, July 2012.

[5] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic CPU-GPU communication management and optimization," *PLDI*, vol. 46, no. 6, p. 142–151, jun 2011. [Online]. Available: https://doi.org/10.1145/1993316.1993516

[6] D. Nuzman, I. Rosen, and A. Zaks, "Auto-vectorization of interleaved data for simd," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06, 2006, p. 132–143. [Online]. Available: https://doi.org/10.1145/1133981.1133997

[7] Y. Chen, C. Mendis, M. Carbin, and S. Amarasinghe, "Vegen: a vectorizer generator for simd and beyond," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 902–914.

[8] M. B. S. Ahmad, A. J. Root, A. Adams, S. Kamil, and A. Cheung, "Vector instruction selection for digital signal processors using program synthesis," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22, 2022, p. 1004–1016. [Online]. Available: https://doi.org/10.1145/3503222.3507714

[9] S. Apostolakis, Z. Xu, Z. Tan, G. Chan, S. Campanoni, and D. I. August, "SCAF: a speculation-aware collaborative dependence analysis framework," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. ACM, 2020. [Online]. Available: https://doi.org/10.1145/3385412.3386028

[10] S. Apostolakis, Z. Xu, G. Chan, S. Campanoni, and D. I. August, "Perspective: A sensible approach to speculative automatic parallelization," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20, 2020. [Online]. Available: https://doi.org/10.1145/3373376.3378458

[11] E. A. Deiana, V. St-Amour, P. A. Dinda, N. Hardavellas, and S. Campanoni, "Unconventional parallelization of nondeterministic applications," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173181

[12] S. Campanoni, G. Holloway, G.-Y. Wei, and D. Brooks, "HELIX-UP: Relaxing program semantics to unleash parallelization," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 235–245. [Online]. Available: http://dl.acm.org/citation.cfm?id=2738600.2738630

[13] S. Campanoni, T. Jones, G. Holloway, G. Y. Wei, and D. Brooks, "The HELIX project: Overview and directions," in *DAC Design Automation Conference 2012*, June 2012, pp. 277–282.

[14] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas, "Posh: a tls compiler that exploits program structure," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2006, pp. 158–167.

[15] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, p. 319–349, jul 1987. [Online]. Available: https://doi.org/10.1145/24039.24041

[16] A. Matni, E. A. Deiana, Y. Su, L. Gross, S. Ghosh, S. Apostolakis, Z. Xu, Z. Tan, I. Chaturvedi, D. I. August, and S. Campanoni, "NOELLE Offers Empowering LLvm Extensions," in *International Symposium on Code Generation and Optimization, 2022. CGO 2022.*, 2022.

[17] B. Guo, M. J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D. I. August, "Practical and accurate low-level pointer analysis," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '05. USA: IEEE Computer Society, 2005. [Online]. Available: https://doi.org/10.1109/CGO.2005.27

[18] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting equality of variables in programs," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '88, 1988. [Online]. Available: https://doi.org/10.1145/73560.73561

[19] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, oct 1991. [Online]. Available: https://doi.org/10.1145/115372.115320

[20] K. Knobe and V. Sarkar, "Array ssa form and its use in parallelization," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '98, 1998. [Online]. Available: https://doi.org/10.1145/268946.268956

[21] S. J. Fink, K. Knobe, and V. Sarkar, "Unified analysis of array and object references in strongly typed languages," in *Proceedings of the 7th International Symposium on Static Analysis*. Berlin, Heidelberg: Springer-Verlag, 2000.

[22] S. H. Yong, S. Horwitz, and T. Reps, "Pointer analysis for programs with structures and casting," *SIGPLAN Not.*, vol. 34, no. 5, may 1999. [Online]. Available: https://doi.org/10.1145/301631.301647

[23] A. Diwan, K. S. McKinley, and J. E. B. Moss, "Type-based alias analysis," *SIGPLAN Not.*, vol. 33, no. 5, may 1998. [Online]. Available: https://doi.org/10.1145/277652.277670

[24] F. Rastello, *SSA-Based Compiler Design*, 1st ed. Springer Publishing Company, Incorporated, 2016.

[25] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004.

[26] R. Stallman, "Using the gnu compiler collection," 01 2004.

[27] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley, "The jalapeño dynamic optimizing compiler for java," in *Proceedings of the ACM 1999 Conference on Java Grande*, ser. JAVA '99, 1999. [Online]. Available: https://doi.org/10.1145/304065.304113

[28] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The tensor algebra compiler," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, oct 2017. [Online]. Available: https://doi.org/10.1145/3133901

[29] S. Chou, F. Kjolstad, and S. Amarasinghe, "Format abstraction for sparse tensor algebra compilers," *Proceedings of the ACM on Programming Languages*, vol. 2, October 2018.

[30] F. Kjolstad, W. Ahrens, S. Kamil, and S. Amarasinghe, "Tensor algebra compilation with workspaces," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2019.

[31] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07, 2007, p. 89–100. [Online]. Available: https://doi.org/10.1145/1250734.1250746

[32] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," in *Proceedings of the 11th International Conference on Compiler Construction*, ser. CC '02. Berlin, Heidelberg: Springer-Verlag, 2002, p. 179–196.

[33] G. Ramalingam, "The undecidability of aliasing," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, sep 1994. [Online]. Available: https://doi.org/10.1145/186025.186041

[34] I. Dillig, T. Dillig, and A. Aiken, "Precise reasoning for programs using containers," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2011. [Online]. Available: https://doi.org/10.1145/1926385.1926407

[35] R. Tarjan, "Testing flow graph reducibility," in *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, ser. STOC '73, 1973, p. 96–107. [Online]. Available: https://doi.org/10.1145/800125.804040

[36] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe, "The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages," *SIGPLAN Not.*, vol. 25, no. 6, jun 1990. [Online]. Available: https://doi.org/10.1145/93548.93578

[37] D. do Couto Teixeira and F. M. Q. Pereira, "The design and implementation of a non-iterative range analysis algorithm on a production compiler," 2011.

[38] V. Paisante, M. Maalej, L. Barbosa, L. Gonnord, and F. M. Quintão Pereira, "Symbolic range analysis of pointers," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2854038.2854050

[39] J. Barnat, P. Bauch, L. Brim, and M. Ceška, "Computing strongly connected components in parallel on cuda," in *2011 IEEE International Parallel and Distributed Processing Symposium*, 2011, pp. 544–555.

[40] J. Cocke, "Global common subexpression elimination," *SIGPLAN Not.*, vol. 5, no. 7, jul 1970. [Online]. Available: https://doi.org/10.1145/390013.808480

[41] G. A. Kildall, "A unified approach to global program optimization," in *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '73. New York, NY, USA: Association for Computing Machinery, 1973. [Online]. Available: https://doi.org/10.1145/512927.512945

[42] L. Ye, M. Lis, and A. Fedorova, "A unifying abstraction for data structure splicing," in *Proceedings of the International Symposium on Memory Systems*, 2019. [Online]. Available: https://doi.org/10.1145/3357526.3357548

[43] T. M. Chilimbi, M. D. Hill, and J. R. Larus, "Cache-conscious structure layout," in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, ser. PLDI '99, 1999. [Online]. Available: https://doi.org/10.1145/301618.301633

[44] S. Rubin, R. Bodík, and T. Chilimbi, "An efficient profile-analysis framework for data-layout optimizations," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002. [Online]. Available: https://doi.org/10.1145/503272.503287

[45] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan, "Classifying memory access patterns for prefetching," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. Lausanne Switzerland: ACM, Mar 2020. [Online]. Available: https://dl.acm.org/doi/10.1145/3373376.3378498

[46] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[47] V. Kiriansky, Y. Zhang, and S. Amarasinghe, "Optimizing indirect memory references with milk," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, ser. PACT '16, 2016, p. 299–312. [Online]. Available: https://doi.org/10.1145/2967938.2967948

[48] D. Racordon, D. Shabalin, D. Zheng, D. Abrahams, and B. Saeta, "Mutable value semantics," *Journal of Object Technology*, vol. 21, 2022. [Online]. Available: http://www.jot.fm/issues/issue_2022_02/article2.pdf

[49] A. Matni, E. A. Deiana, Y. Su, L. Gross, S. Ghosh, S. Apostolakis, Z. Xu, Z. Tan, I. Chaturvedi, D. I. August, and S. Campanoni, "NOELLE Offers Empowering LLvm Extensions," 2021.

[50] V. Sarkar and K. Knobe, "Enabling sparse constant propagation of array elements via array ssa form," in *Static Analysis*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998.

[51] S. Rus, G. He, C. Alias, and L. Rauchwerger, "Region array ssa," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '06, 2006. [Online]. Available: https://doi.org/10.1145/1152154.1152165

[52] R. Surendran, R. Barik, J. Zhao, and V. Sarkar, "Inter-iteration scalar replacement using array ssa form," in *Compiler Construction*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2014.

[53] R. Bodík, R. Gupta, and V. Sarkar, "Abcd: Eliminating array bounds checks on demand," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, 2000. [Online]. Available: https://doi.org/10.1145/349299.349342

[54] D. Novillo *et al.*, "Memory ssa-a unified approach for sparsely representing memory operations," 2007.

[55] Y. Sui, H. Yan, Z. Zheng, Y. Zhang, and J. Xue, "Parallel construction of interprocedural memory ssa form," *Journal of Systems and Software*, vol. 146, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016412121830205X

[56] F. Chow, S. Chan, S. M. Liu, R. Lo, and M. Streich, *Effective representation of aliases and indirect memory operations in SSA form*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, vol. 1060. [Online]. Available: http://link.springer.com/10.1007/3-540-61053-7_66

[57] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14.

[58] T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler, "Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356173

[59] M. Kotsifakou, P. Srivastava, M. D. Sinclair, R. Komuravelli, V. Adve, and S. Adve, "Hpvm: Heterogeneous parallel virtual machine," *SIGPLAN Not.*, vol. 53, no. 1, p. 68–80, feb 2018. [Online]. Available: https://doi.org/10.1145/3200691.3178493

[60] G. Chaitin, "Register allocation and spilling via graph coloring," *SIGPLAN Not.*, vol. 39, no. 4, apr 2004. [Online]. Available: https://doi.org/10.1145/989393.989403

[61] F. C. Chow and J. L. Hennessy, "The priority-based coloring approach to register allocation," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 4, oct 1990. [Online]. Available: https://doi.org/10.1145/88616.88621

[62] P. Briggs, K. D. Cooper, and L. Torczon, "Improvements to graph coloring register allocation," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, may 1994. [Online]. Available: https://doi.org/10.1145/177492.177575

[63] P. Bergner, P. Dahl, D. Engebretsen, and M. O'Keefe, "Spill code minimization via interference region spilling," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, ser. PLDI '97, 1997. [Online]. Available: https://doi.org/10.1145/258915.258941

[64] S. Hack, D. Grund, and G. Goos, *Register Allocation for Programs in SSA-Form*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, vol. 3923. [Online]. Available: http://link.springer.com/10.1007/11688839_20

[65] Y. G. Park and B. Goldberg, "Escape analysis on lists," in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, 1992. [Online]. Available: https://doi.org/10.1145/143095.143125

[66] B. Blanchet, "Escape analysis for object-oriented languages: Application to java," in *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1999. [Online]. Available: https://doi.org/10.1145/320384.320387

[67] D. Gay and B. Steensgaard, "Fast escape analysis and stack allocation for object-based programs," in *Proceedings of the 9th International Conference on Compiler Construction*, 2000.

[68] G. M. Essertel, G. Wei, and T. Rompf, "Precise reasoning with structured time, structured heaps, and collective operations," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019. [Online]. Available: https://doi.org/10.1145/3360583

[69] [Online]. Available: https://www.swift.org/

[70] [Online]. Available: https://www.hylo-lang.org/

[71] N. D. Matsakis and F. S. Klock, "The rust language," *Ada Lett.*, vol. 34, no. 3, p. 103–104, oct 2014. [Online]. Available: https://doi.org/10.1145/2692956.2663188

[72] [Online]. Available: https://docs.modular.com/mojo/manual/values/ownership.html

[73] [Online]. Available: https://vale.dev/

[74] K. E. Iverson, *A Programming Language*. USA: John Wiley & Sons, Inc., 1962.

[75] M. C. Harrison, "Balm: An extendable list-processing language," in *Proceedings of the May 5-7, 1970, Spring Joint Computer Conference*, ser. AFIPS '70 (Spring), 1970, p. 507–511. [Online]. Available: https://doi.org/10.1145/1476936.1477015

[76] R. E. Griswold, *A History of the SNOBOL Programming Languages*, 1978, p. 601–645. [Online]. Available: https://doi.org/10.1145/800025.1198417

[77] J. T. Schwartz, R. B. Dewar, E. Schonberg, and E. Dubinsky, *Programming with Sets; an Introduction to SETL*. Berlin, Heidelberg: Springer-Verlag, 1986.

[78] J. T. Schwartz, "Automatic and semiautomatic optimization of setl," in *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, 1974, p. 43–49. [Online]. Available: https://doi.org/10.1145/800233.807044

[79] S. Westrick, M. Rainey, D. Anderson, and G. E. Blelloch, "Parallel block-delayed sequences," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022. [Online]. Available: https://doi.org/10.1145/3503221.3508434