Representing Data Collections in an SSA Form

Tommy McMichen

CGO 2024

Northwestern University

ARCANA
Rethinks Compiler Abstractions for New Applications

CONSTELLATION PROJECT

m&moir

Outline

In today's talk:

Motivations

How *Data Collections* block compilers.

Outline

In today's talk:

Motivations

How *Data Collections* block compilers.

Proposal

Introduce MEMOIR: the *first general-purpose SSA IR for Data Collections.*

Outline

In today's talk:

Motivations

How *Data Collections* block compilers.

Proposal

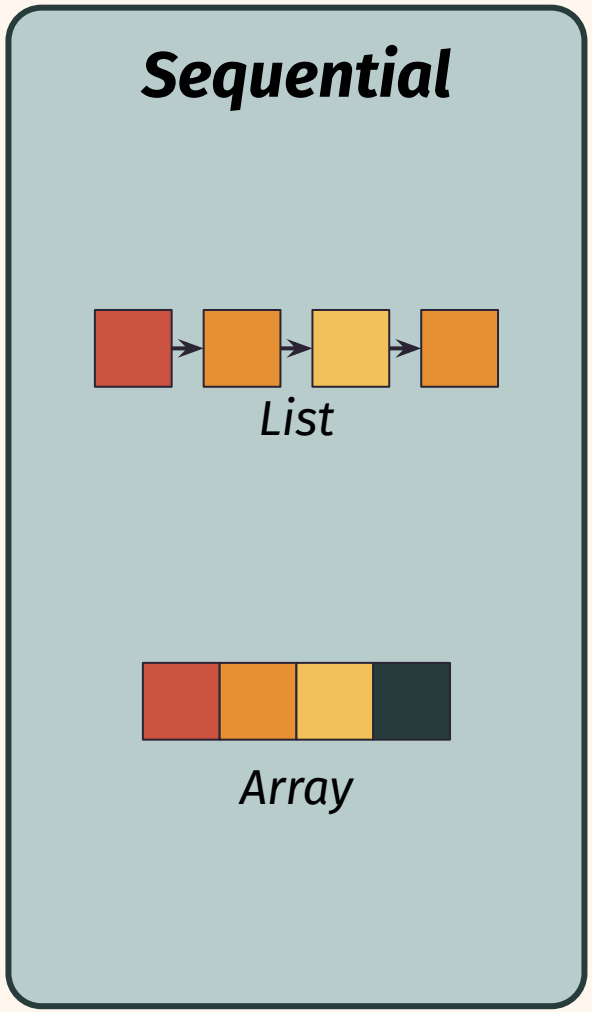
Introduce MEMOIR: the *first general-purpose SSA IR for Data Collections.*

Evaluation

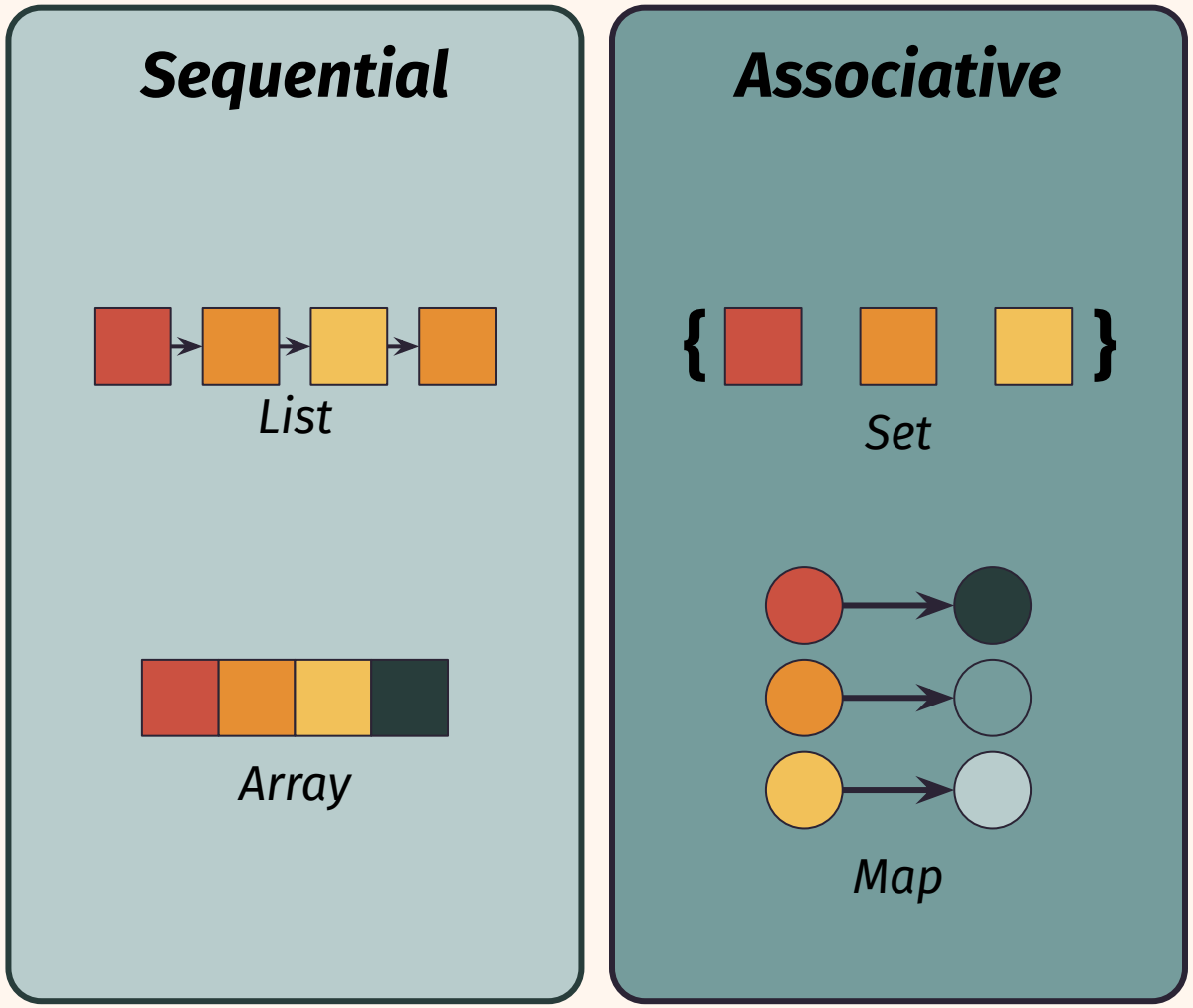
Demonstrate optimizations that are now possible with MEMOIR.

Data Collection:
A logical organization of data.

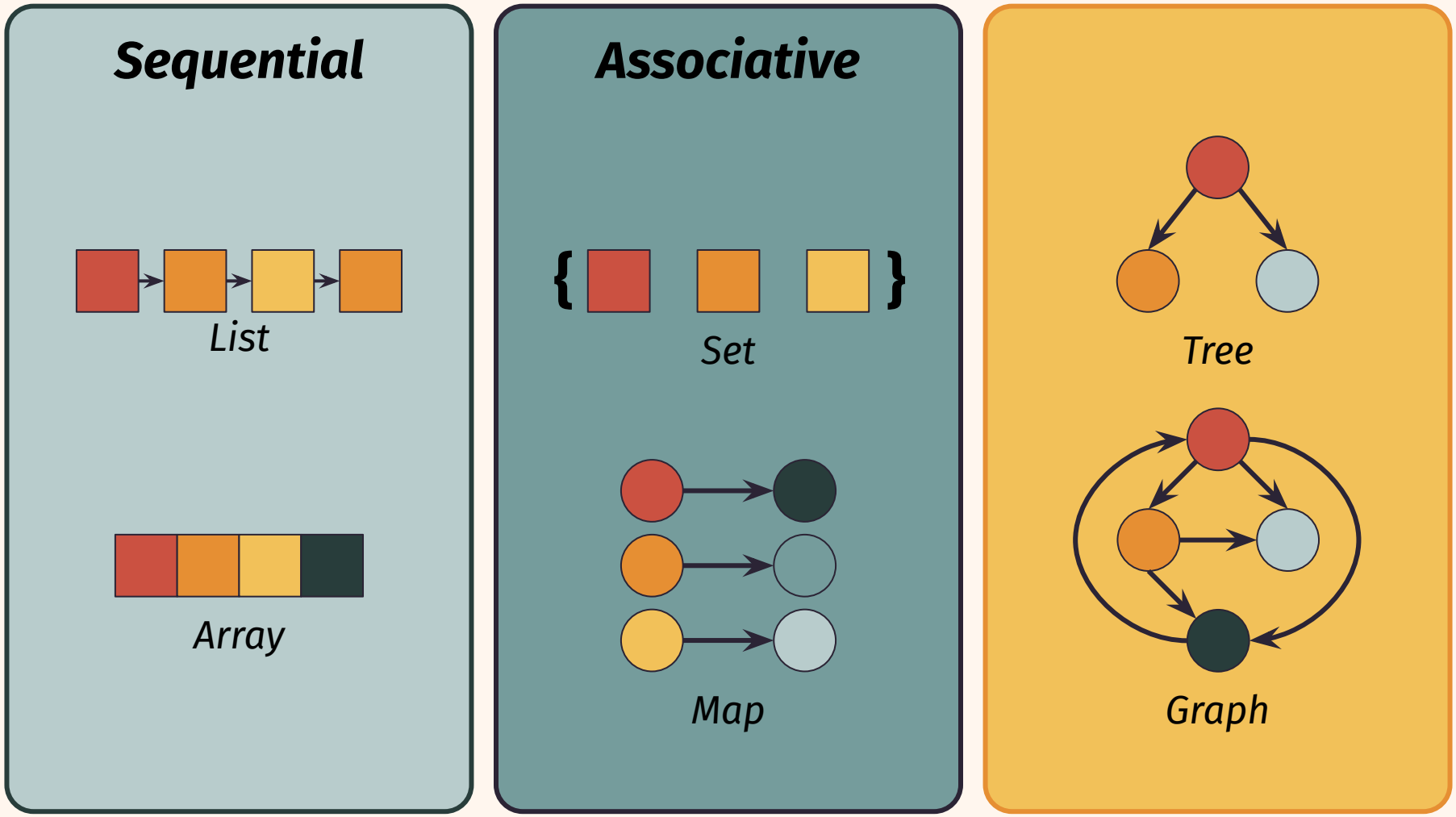
Data Collection: ***A logical organization of data.***



Data Collection: ***A logical organization of data.***



Data Collection: ***A logical organization of data.***



Motivations

Most Accesses to Heap Memory are for Collections

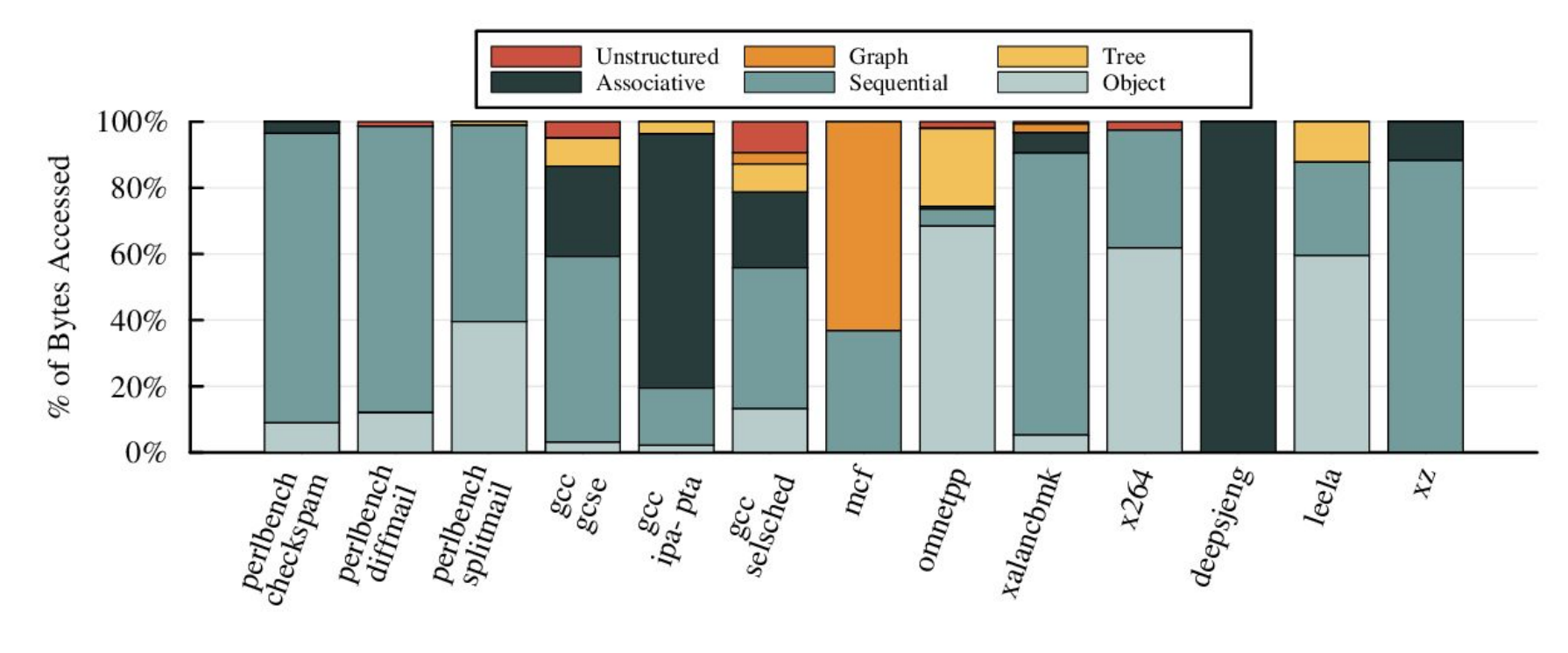


Figure: Breakdown of bytes read and written for each memory class in SPECINT 2017.

Motivations

Most Accesses to Heap Memory are for Structured Data

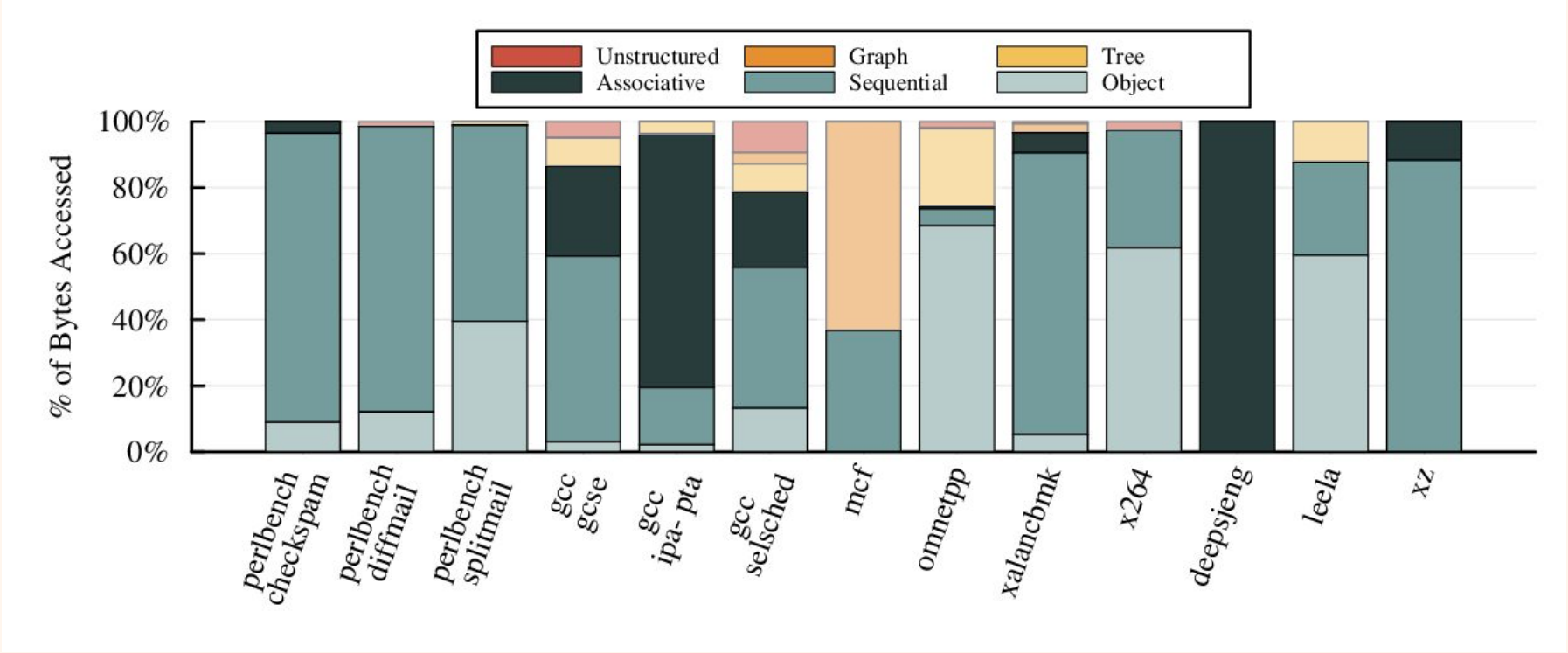


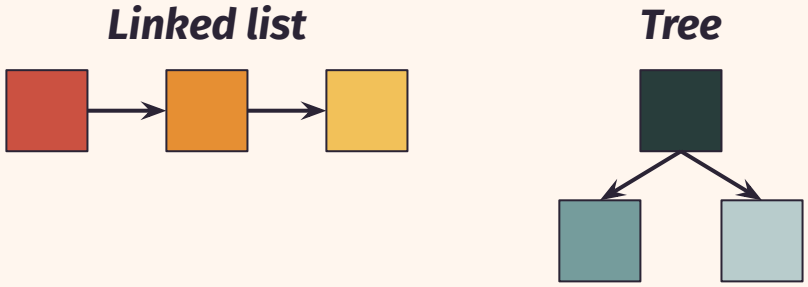
Figure: Breakdown of bytes read and written for each memory class in SPECINT 2017.

The Problem

Data collection implementations introduce complex memory behavior.

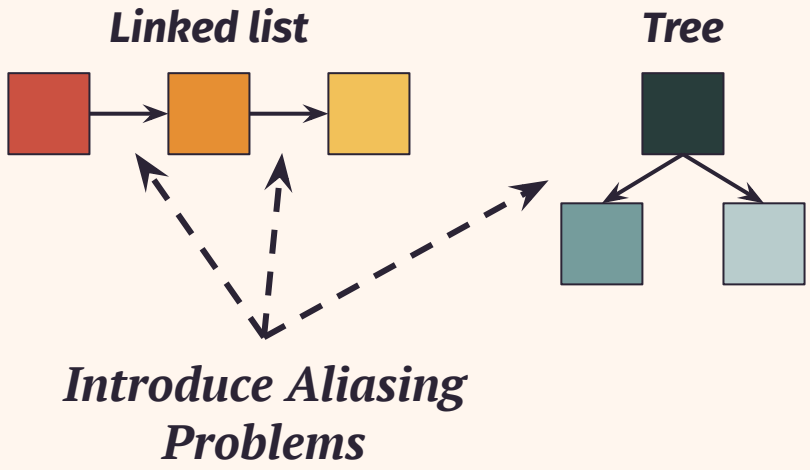
Complex Memory Behavior

Linked data structures



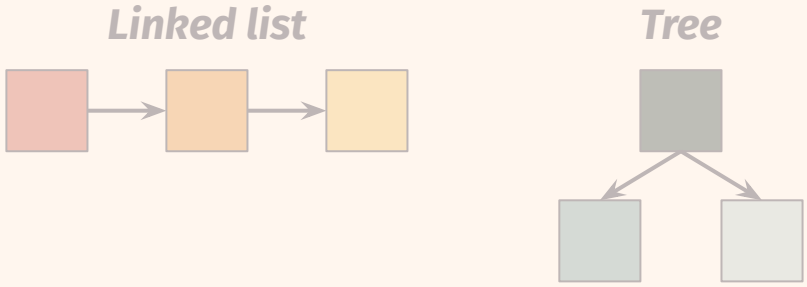
Complex Memory Behavior

Linked data structures

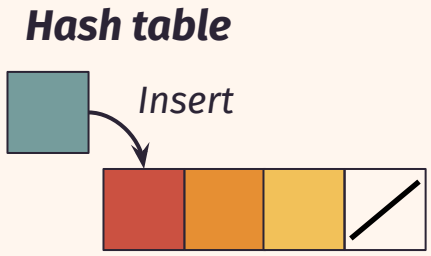


Complex Memory Behavior

Linked data structures

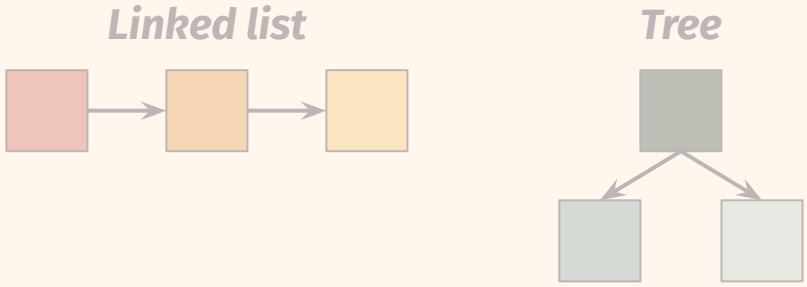


Expanding data structures

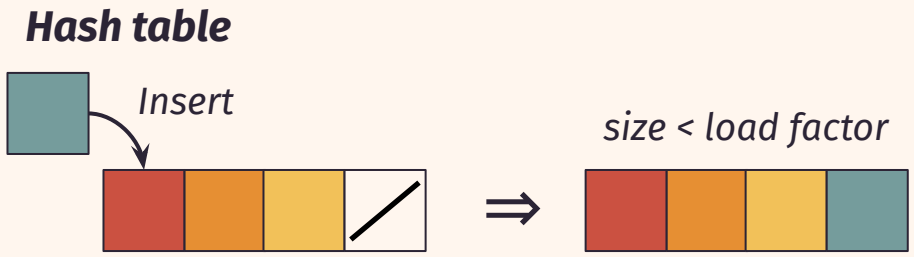


Complex Memory Behavior

Linked data structures

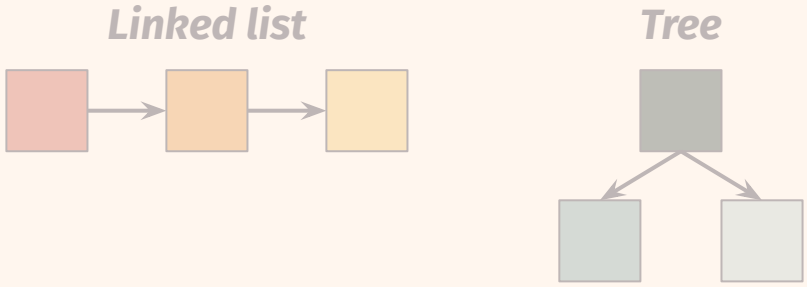


Expanding data structures

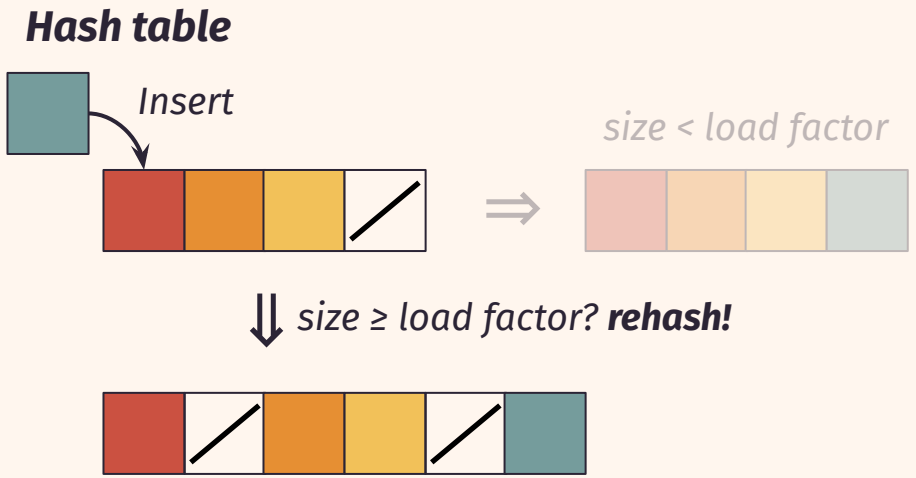


Complex Memory Behavior

Linked data structures

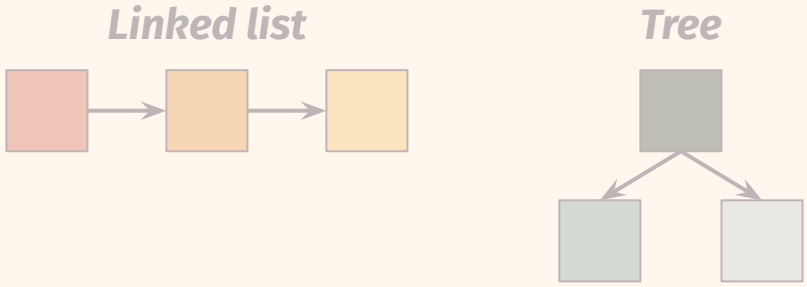


Expanding data structures

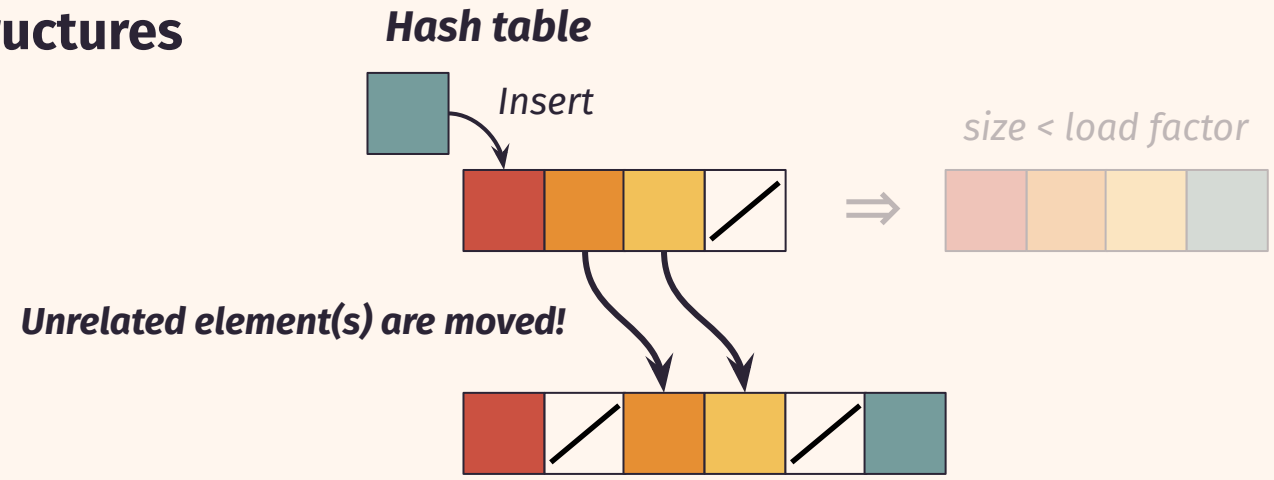


Complex Memory Behavior

Linked data structures



Expanding data structures



Expanding Data Structures *block optimizations!*

```
std::unordered_map<int, int> table = ...;

table[0] = 10;
table[1] = 20;
print(table[0]);
```

Expanding Data Structures *block optimizations!*

```
std::unordered_map<int, int> table = ...;

table[0] = 10;
table[1] = 20;
print(table[0]);
```

Expanding Data Structures *block optimizations!*

```
std::unordered_map<int, int> table = ...;

table[0] = 10;
table[1] = 20;
print(table[0]);
```

No production compiler can propagate **10** to the `print` statement

Expanding Data Structures *block optimizations!*

```
std::unordered_map<int, int> table = ...;

table[0] = 10;
table[1] = 20;  ──→ { realloc(table, ...);
print(table[0]);      rehash(table);
```

Simple operations → complex memory behavior.

Expanding Data Structures *block optimizations!*

```
std::unordered_map<int, int> table = ...;

table[0] = 10;
table[1] = 20;
print(table[0]);
```

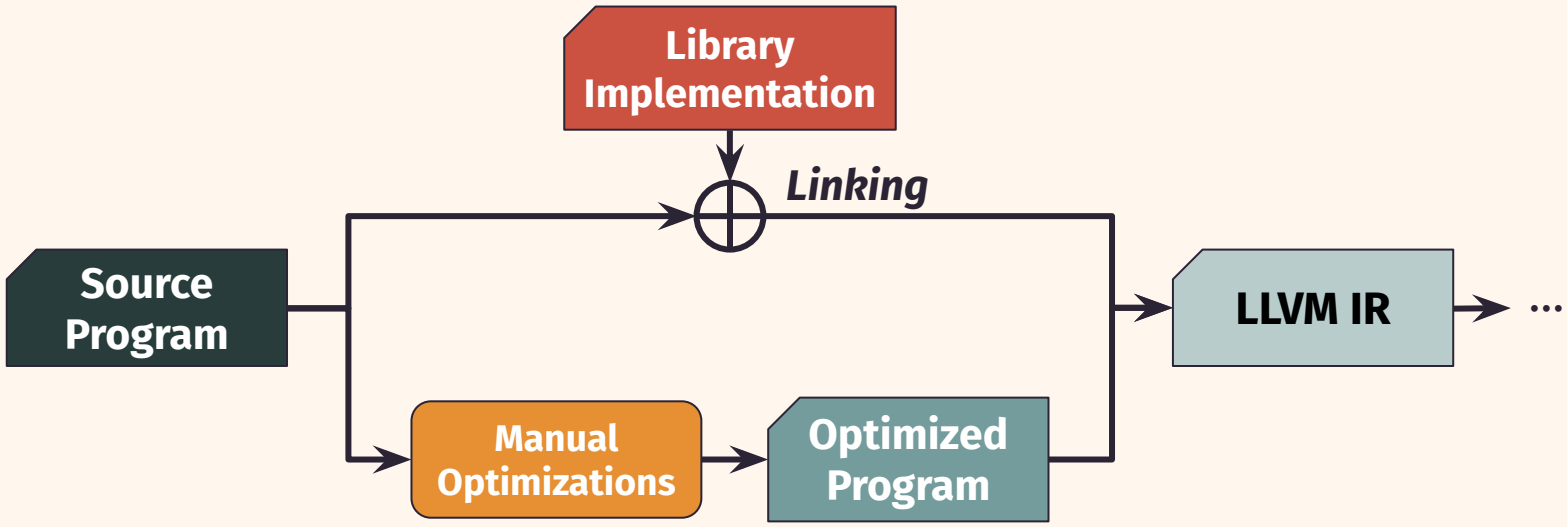
$\neq$

```
realloc(table, ...);
rehash(table);
```

Complex memory behavior blocks optimizations!

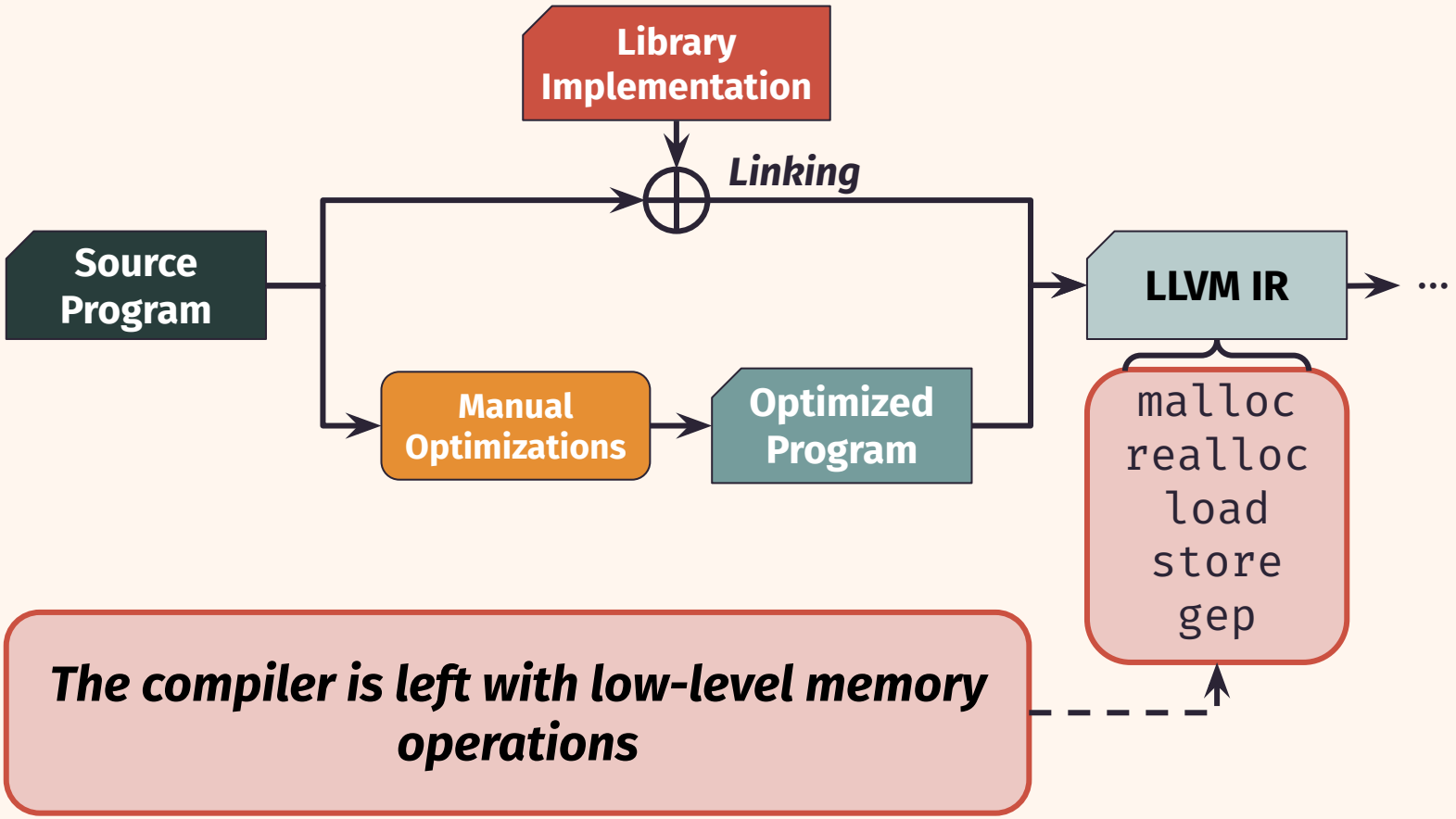
Insights

Stems from *premature lowering* to *fixed implementations* manually or via libraries



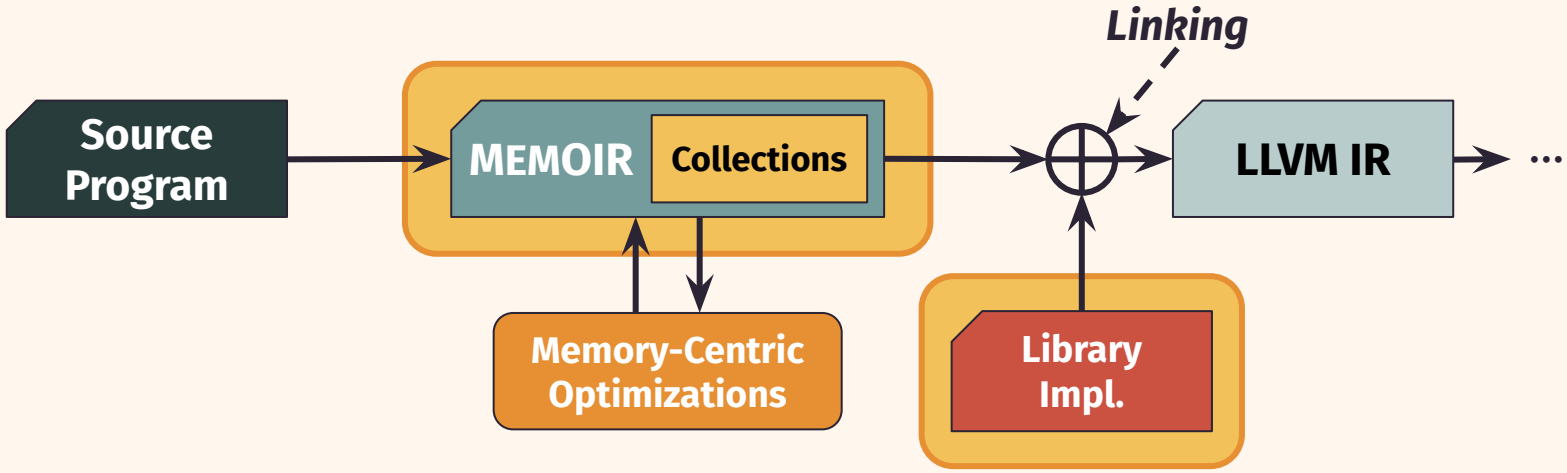
Insights

Stems from *premature lowering* to *fixed implementations* manually or via libraries



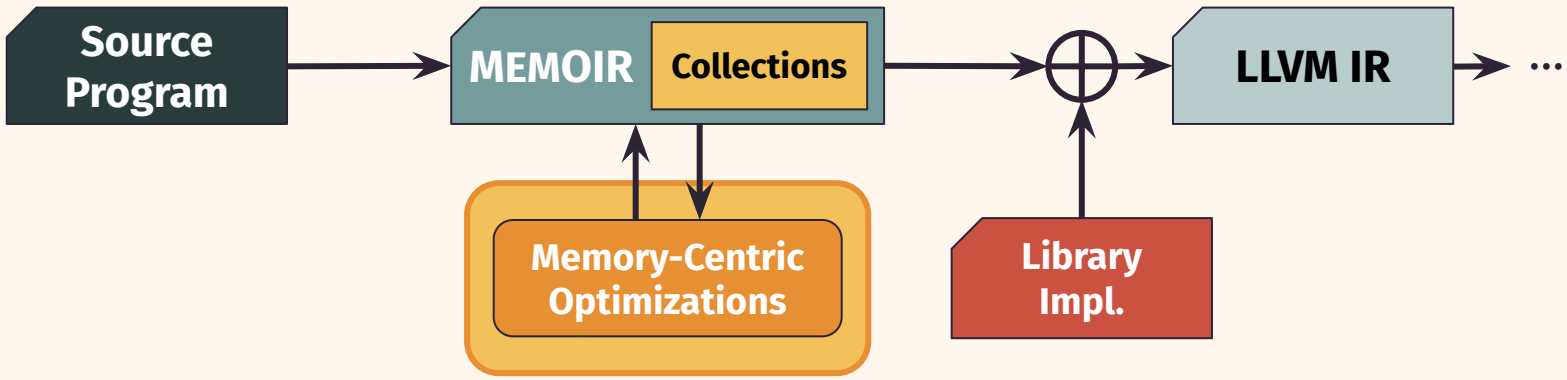
Proposal

Progressively lower to MEMOIR *before* library implementation



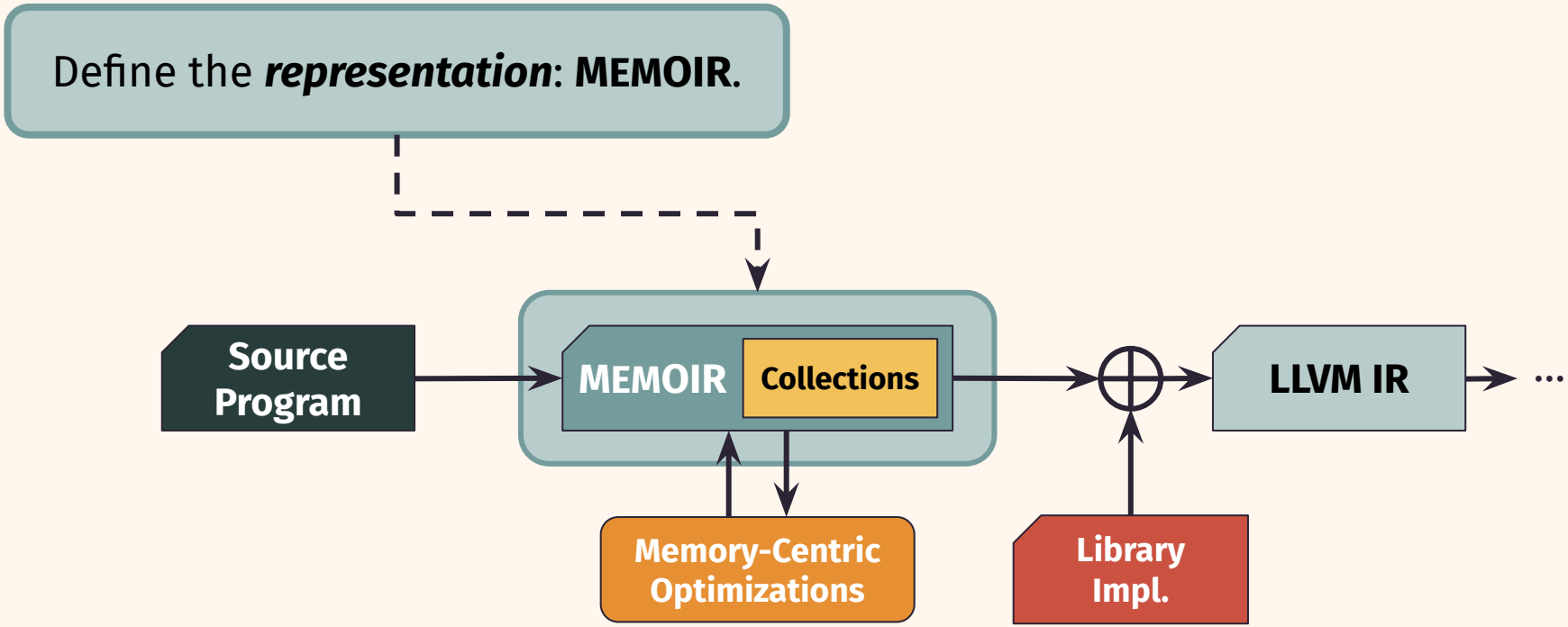
Proposal

Implement *Memory Optimizations* within the Compiler for *Easy, Automatic Reuse*



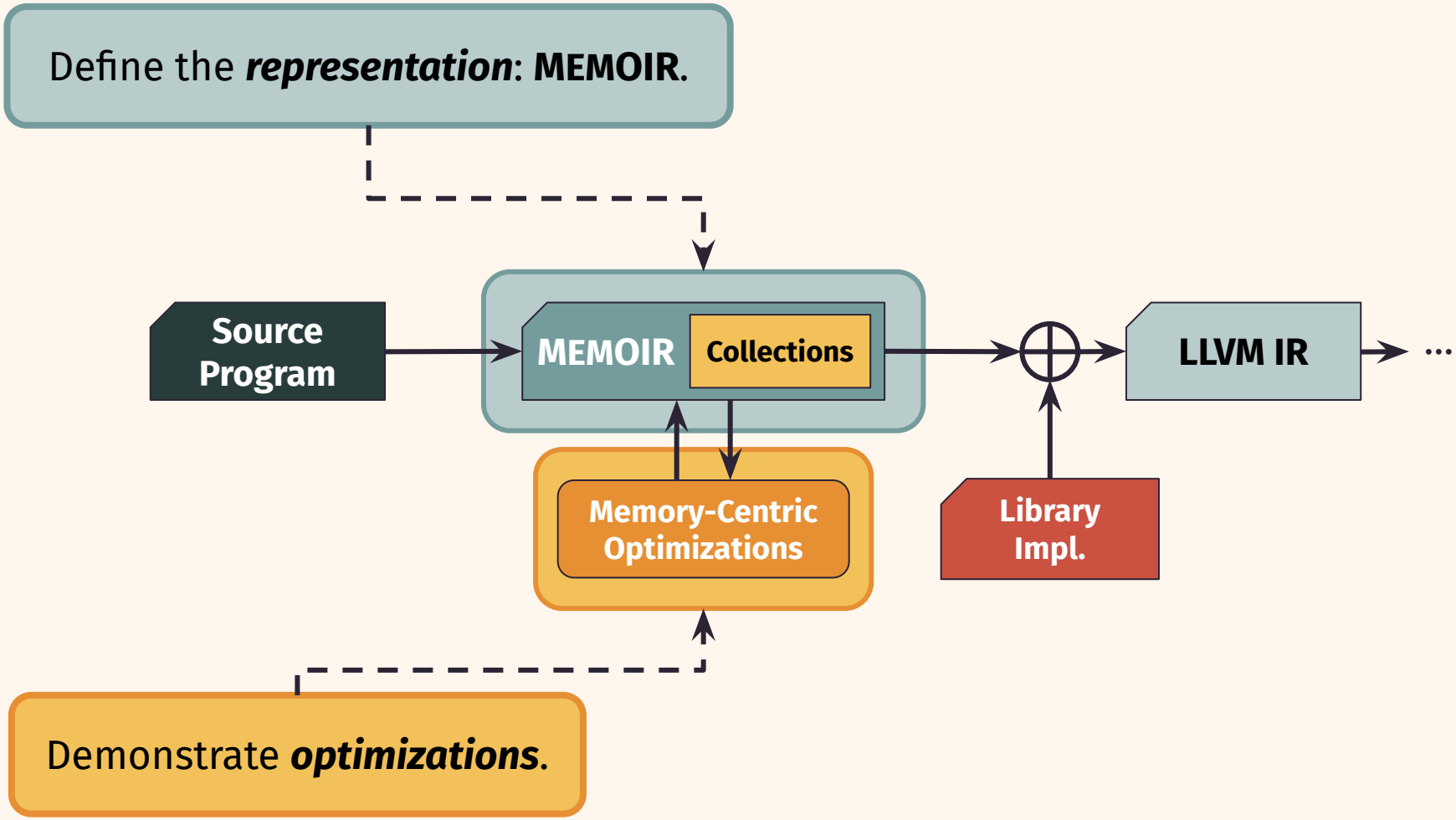
Proposal

In the paper:



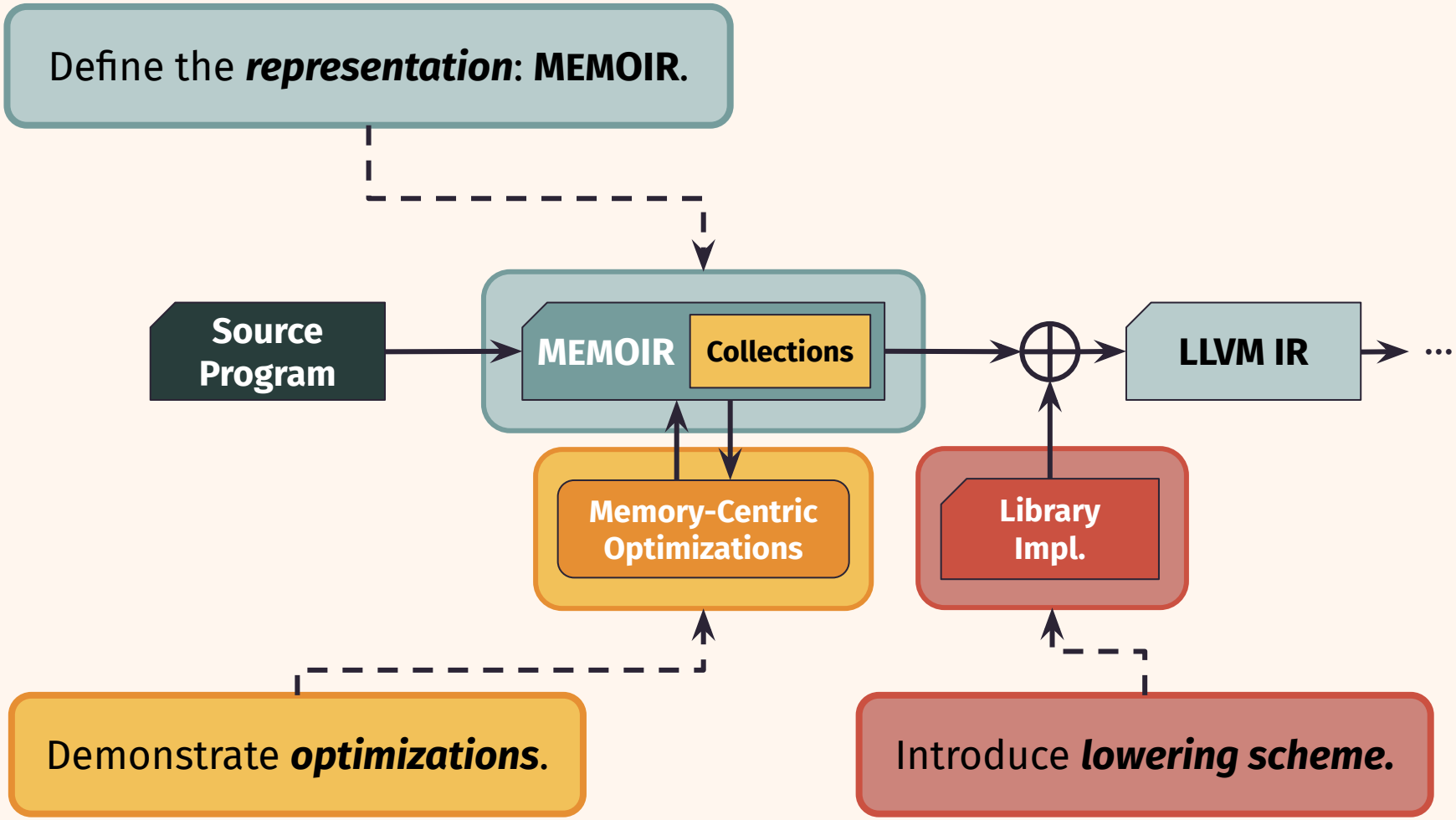
Proposal

In the paper:



Proposal

In the paper:



MEMOIR

Representing data collections for analysis and transformation

Goal: Provide a compiler intermediate representation with...

Unambiguous memory operations

Goal: Provide a compiler intermediate representation with...

Unambiguous memory operations

Element-level analyzability

Goal: Provide a compiler intermediate representation with...

Unambiguous memory operations

Element-level analyzability

Ability to ***transform memory layout*** of *collections* and *single objects*

Our Approach

Unambiguous memory operations

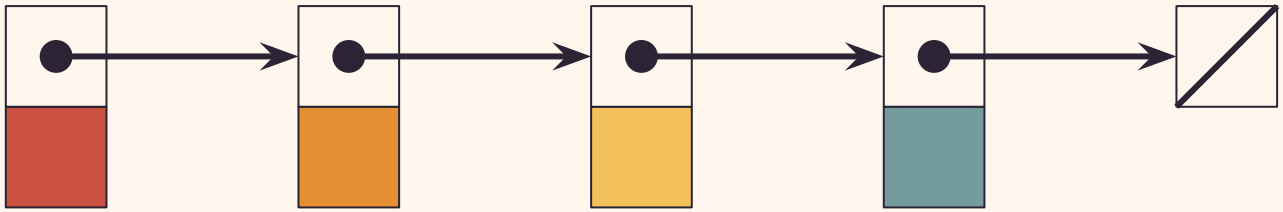
Element-level analyzability

Ability to ***transform memory layout*** of *collections* and *single objects*

Achieved by ***decoupling*** memory used to ***store*** *data* from memory used to ***logically organize*** *data*.

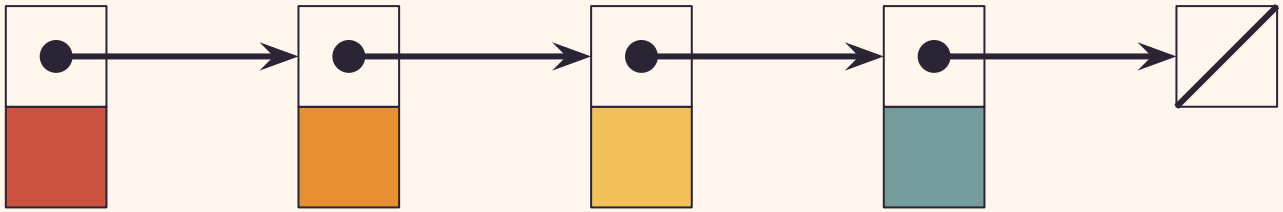
Decoupling Data from its Logical Organization

Example: Linked List



Decoupling Data from its Logical Organization

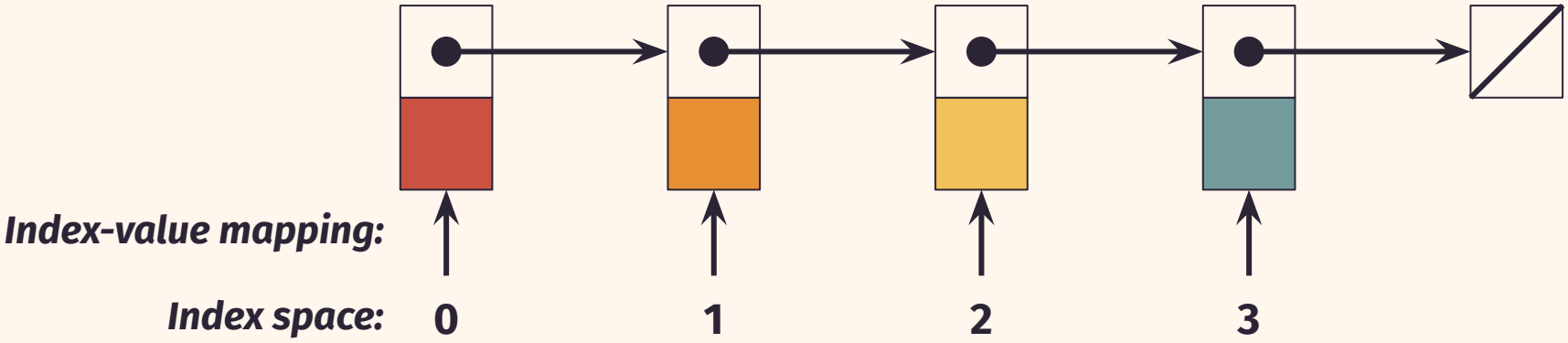
Example: Linked List



Index space: **0** **1** **2** **3**

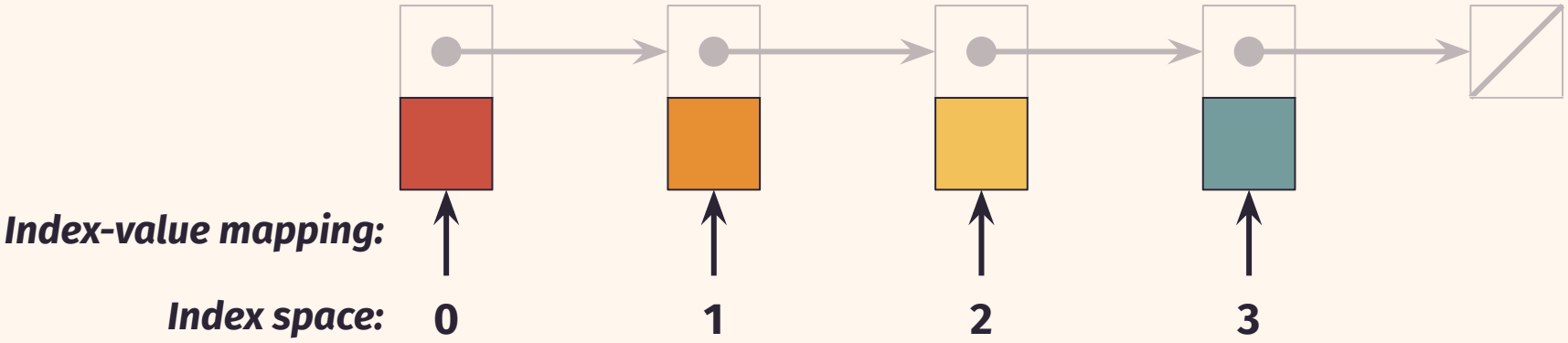
Decoupling Data from its Logical Organization

Example: Linked List



Decoupling Data from its Logical Organization

Example: Linked List



Abstract away the memory used to ***logically organize*** the collection.

Capture two common cases:

Associative

Uniqueness in index space

Capture two common cases:

Associative
Uniqueness in index space

Sequential
Contiguous in index space

MemOIR

Element-Level Analysis

With this decoupling, we can analyze collections at the ***granularity of elements.***

Where *scalar* analysis and transformation *fails*.

LLVM IR, etc.

Constant *Scalar* Propagation

```
... table = ... ;

table[0] = 10;
table[1] = 20;
print(table[0]);
```

Element-Level analysis and transformation can *prevail*.

LLVM IR, etc.

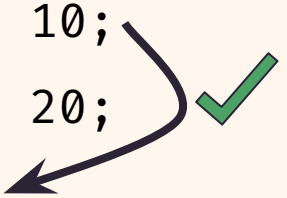
Constant *Scalar* Propagation

```
... table = ... ;

table[0] = 10;
table[1] = 20;
print(table[0]);
```

MEMOIR

Constant *Element* Propagation

```
... table = ... ;

table[0] = 10;
table[1] = 20;
print(10);
```



Construct a MEMOIR Program

LLVM IR, etc. **MEMOIR**

Constant *Scalar* Propagation

Constant *Element* Propagation

```
... table = ... ;  ⟶  Assoc<int,int> table = ... ;
```

```
table[0] = 10;
table[1] = 20;
print(table[0]);
```

Construct a MEMOIR Program

LLVM IR, etc. **MEMOIR**

Constant *Scalar* Propagation

Constant *Element* Propagation

```
... table = ... ;  ⟶  Assoc<int,int> table = ... ;

table[0] = 10;     ⟶  table1 = INSERT(table, 0, 10);
table[1] = 20;
print(table[0]);
```

Construct a MEMOIR Program

LLVM IR, etc. **MEMOIR**

Constant *Scalar* Propagation

Constant *Element* Propagation

```
... table = ... ;  ⟶  Assoc<int,int> table = ... ;

table[0] = 10;     ⟶  table1 = INSERT(table, 0, 10);
table[1] = 20;
print(table[0]);
```

Construct a MEMOIR Program

LLVM IR, etc. **MEMOIR**

Constant *Scalar* Propagation

Constant *Element* Propagation

```
... table = ... ;  ⟶  Assoc<int,int> table = ... ;

table[0] = 10;     ⟶  table1 = INSERT(table,  0, 10);
table[1] = 20;     ⟶  table2 = INSERT(table1, 1, 20);
print(table[0]);
```

Construct a MEMOIR Program

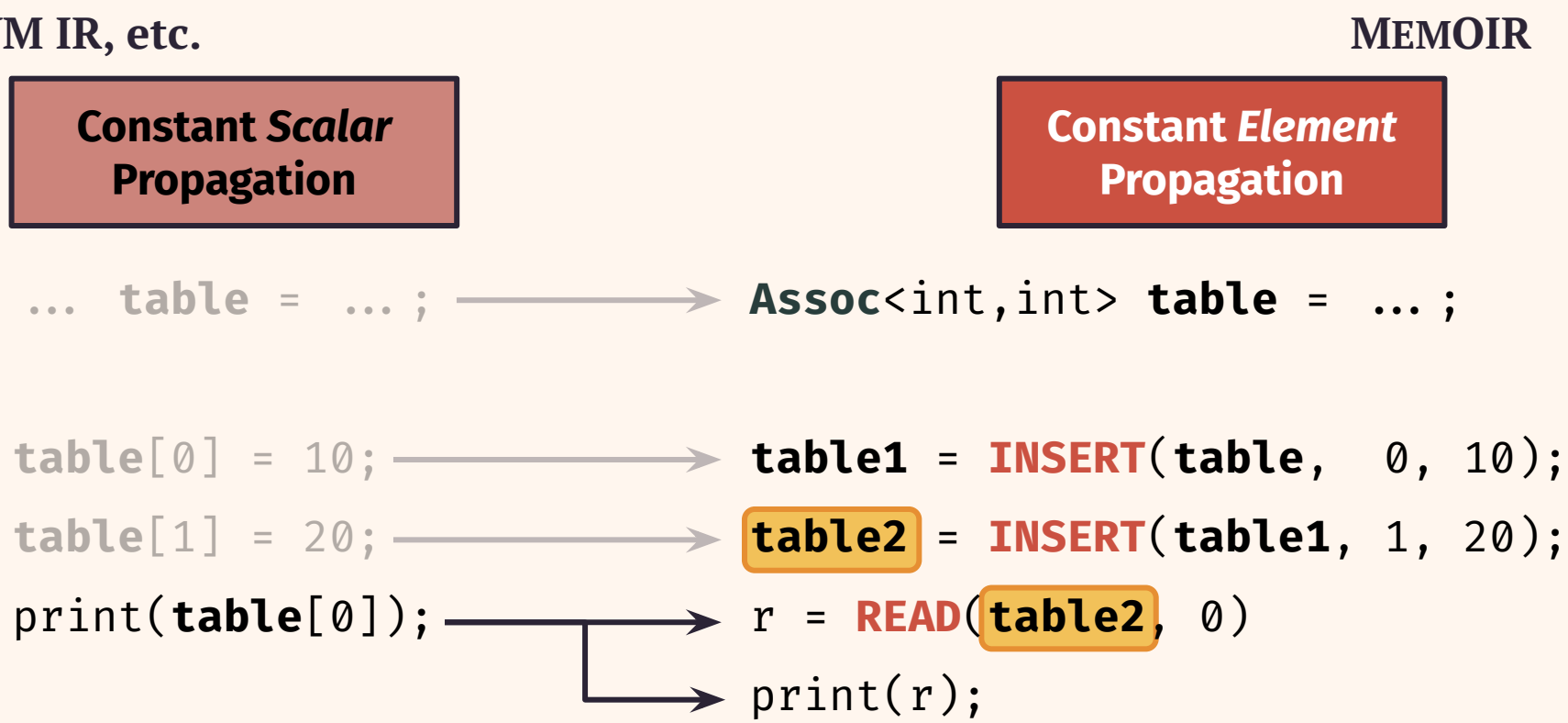
LLVM IR, etc. **MEMOIR**

Constant *Scalar* Propagation

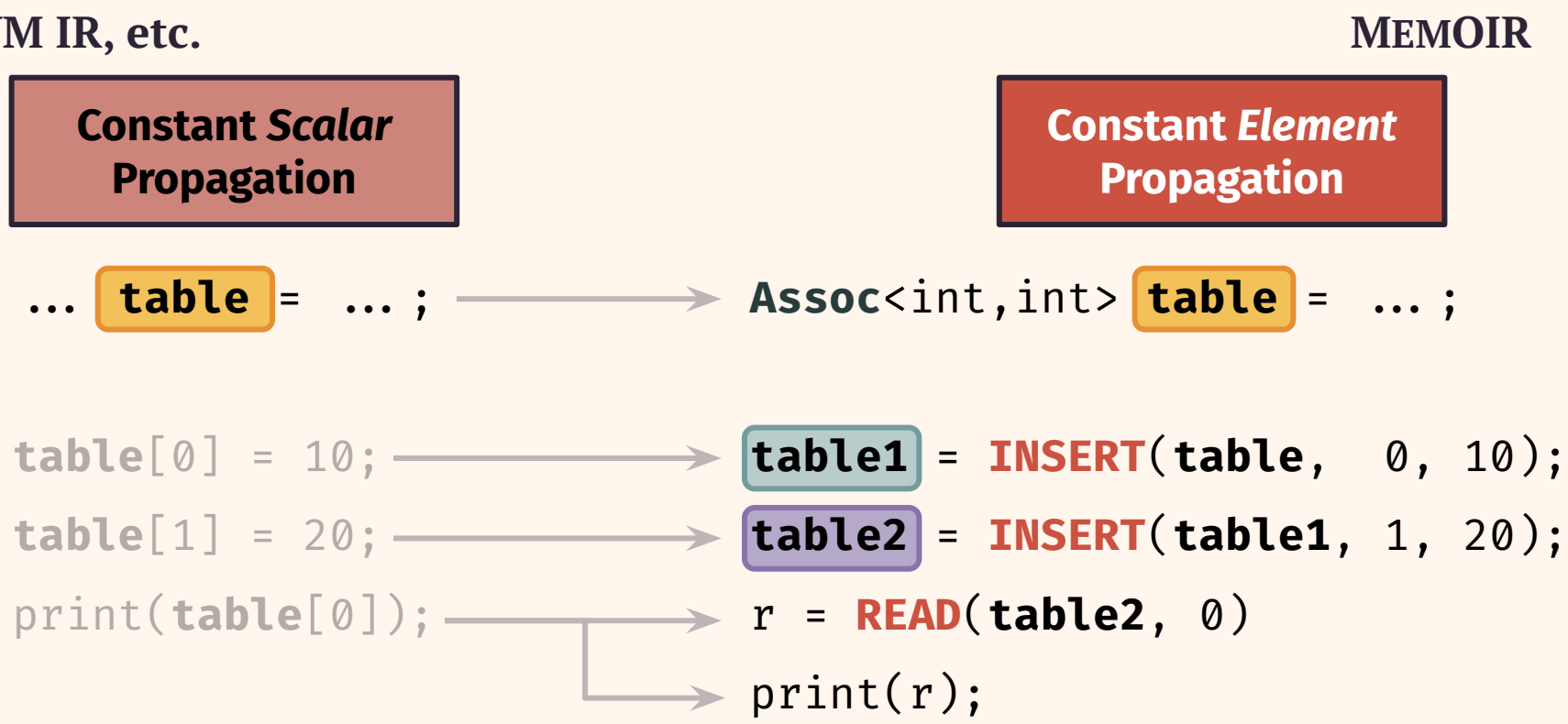
Constant *Element* Propagation

```
... table = ... ;          ⟶   Assoc<int,int> table = ... ;

table[0] = 10;             ⟶   table1 = INSERT(table,  0, 10);
table[1] = 20;             ⟶   table2 = INSERT(table1, 1, 20);
print(table[0]);
```

Construct a MEMOIR Program



Construct a MEMOIR Program



Construct a MEMOIR Program

Benchmark	Number of Collections	
	Source	SSA
`mcf`	5	13
`deepsjeng`	2	14
`LLVM opt`	8	37

SSA Construction introduces new collections

Construct a MEMOIR Program

Benchmark	Number of Collections		
	Source	SSA	Binary
`mcf`	5	13 →	5
`deepsjeng`	2	14 →	2
`LLVM opt`	8	37 →	8

But SSA Destruction introduces *no spurious copies!*

Performing a *sparse data flow analysis* on collections.

LLVM IR, etc.

Constant *Scalar* Propagation

MEMOIR

Constant *Element* Propagation

```
{ }                ◁ Assoc<int,int> table = ... ;

{ 0→10 }           ◁ table1 = INSERT(table,  0, 10);
{ 0→10, 1→20 }     ◁ table2 = INSERT(table1, 1, 20);
                     r = READ(table2, 0)
                     print(r);
```

Propagate *element-level* constants to optimize the program.

LLVM IR, etc.

Constant *Scalar* Propagation

MEMOIR

Constant *Element* Propagation

```
{ }          ◁  Assoc<int,int> table =  ... ;

{ 0→10 }     ◁  table1 = INSERT(table,  0, 10);
{ 0→10, 1→20 } ◁  table2 = INSERT(table1, 1, 20);
                  r = READ(table2, 0)
                  print(r);
```

Propagate *element-level* constants to optimize the program.

LLVM IR, etc.

Constant *Scalar* Propagation

MEMOIR

Constant *Element* Propagation

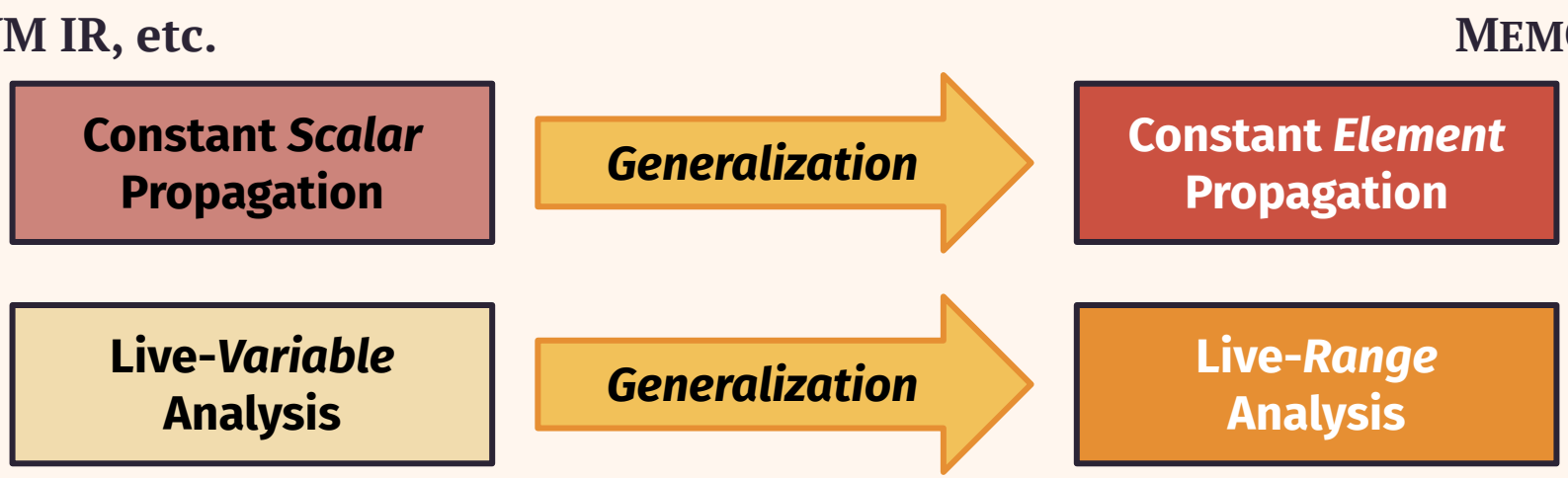
```
{ }  ◁  Assoc<int,int> table = ... ;

{ 0→10 }  ◁  table1 = INSERT(table,  0, 10);
{ 0→10, 1→20 }  ◁  table2 = INSERT(table1, 1, 20);
                   r = 10;
                   print(r);
```

Generalizing scalar optimizations to operate on *collections* and single *objects* with MEMOIR

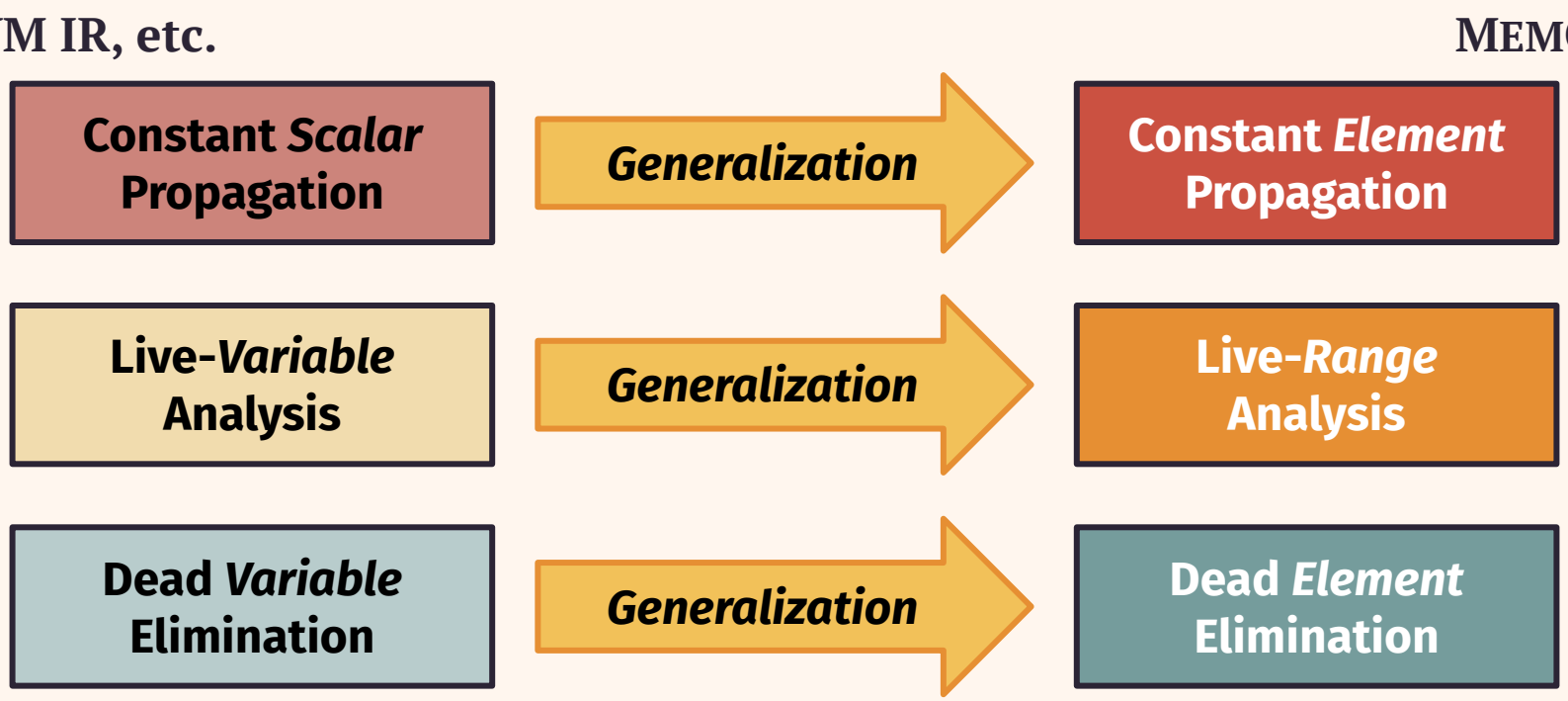
LLVM IR, etc.

MEMOIR

Generalizing scalar optimizations to operate on *collections* and single *objects* with MEMOIR

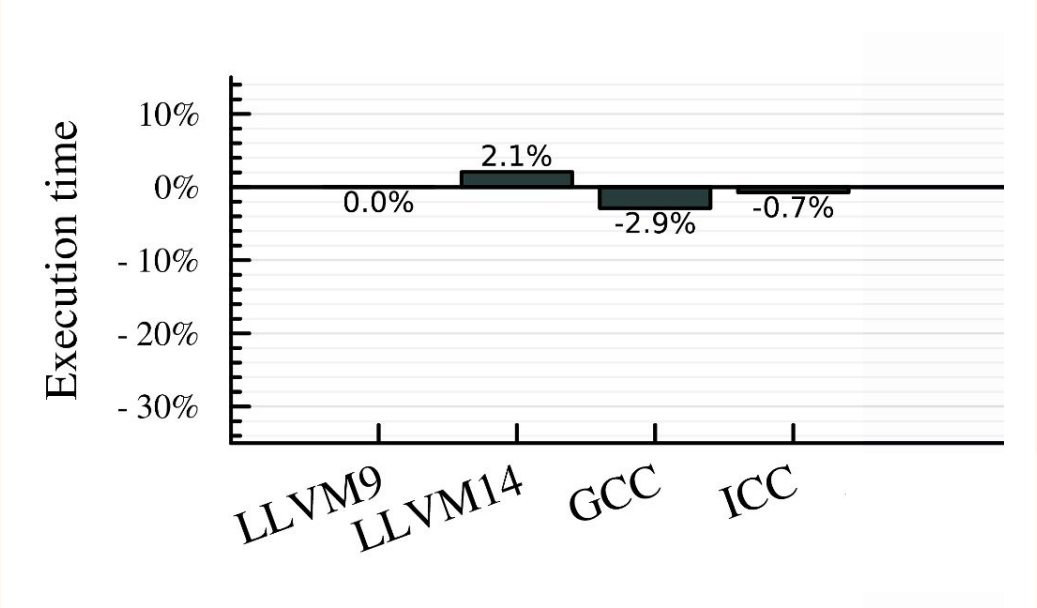


Generalizing scalar optimizations to operate on *collections* and single *objects* with MEMOIR



Evaluation

Production compilers provide negligible performance improvements on mcf_s.



Lower is better.

Figure: Execution time of `mcf_s` with `refspeed` input. 10 trials. Normalized to `LLVM9`.

Evaluation

MEMOIR provides significant performance improvements with several optimizations.

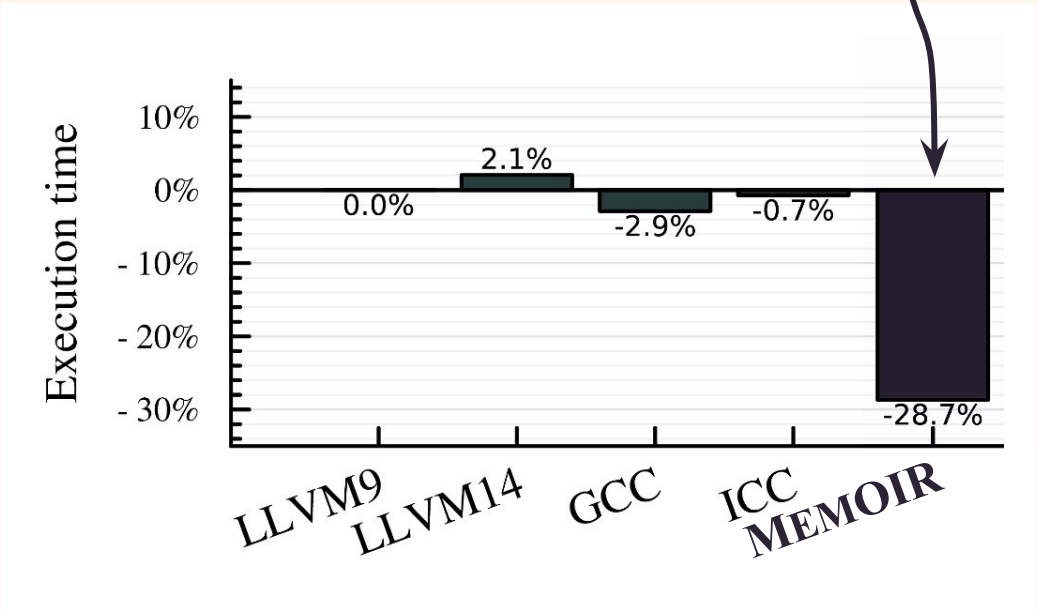


Figure: Execution time of `mcf_s` with `refspeed` input. 10 trials. Normalized to `LLVM9`.

Example Application: mcf_s from SPEC2017

Quick sort accounts for ***~40%*** of exec. time

```
Seq<T> sorted = qsort(in);

for (i = 0 to K)
    r = READ(sorted, i);
    if (r > threshold)
        use(r);
```

Evaluation

Live Range Analysis propagates liveness information

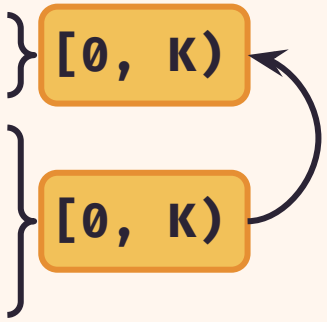
```
Seq<T> sorted = qsort(in);

for (i = 0 to K)
    r = READ(sorted, i);
    if (r > threshold)
        use(r);
```

`[0, end)`

`[0, K)`

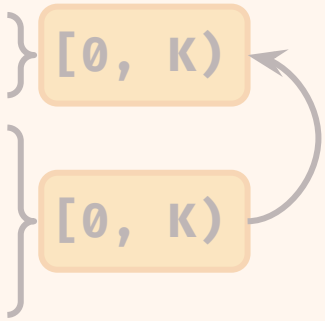
Live Range Analysis propagates liveness information

```
Seq<T> sorted = qsort(in);

for (i = 0 to K)
    r = READ(sorted, i);
    if (r > threshold)
        use(r);
```



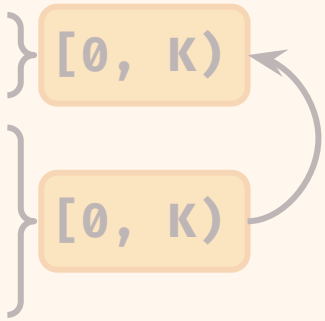
Dead Element Elimination converts sort → partial sort!

```
Seq<T> sorted = qsort'(in, 0, K);

for (i = 0 to K)
    r = READ(sorted, i);
    if (r > threshold)
        use(r);
```



Dead Element Elimination converts sort → partial sort!

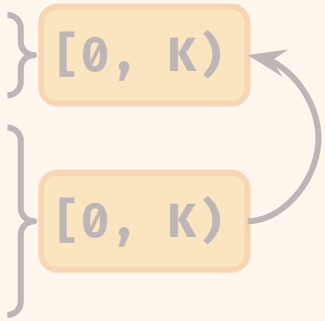
```
Seq<T> sorted = qsort'(in, 0, K);

for (i = 0 to K)
    r = READ(sorted, i);
    if (r > threshold)
        use(r);
```



With no primitive knowledge of sort!

Dead Element Elimination converts sort → partial sort!

```
Seq<T> sorted = qsort'(in, 0, K);

for (i = 0 to K)
    r = READ(sorted, i);
    if (r > threshold)
        use(r);
```



With no primitive knowledge of sort!

See the paper for more details.

Evaluation

MEMOIR reduces *O(nlogn)* operation to *O(n+klogk)*, *where k << n*

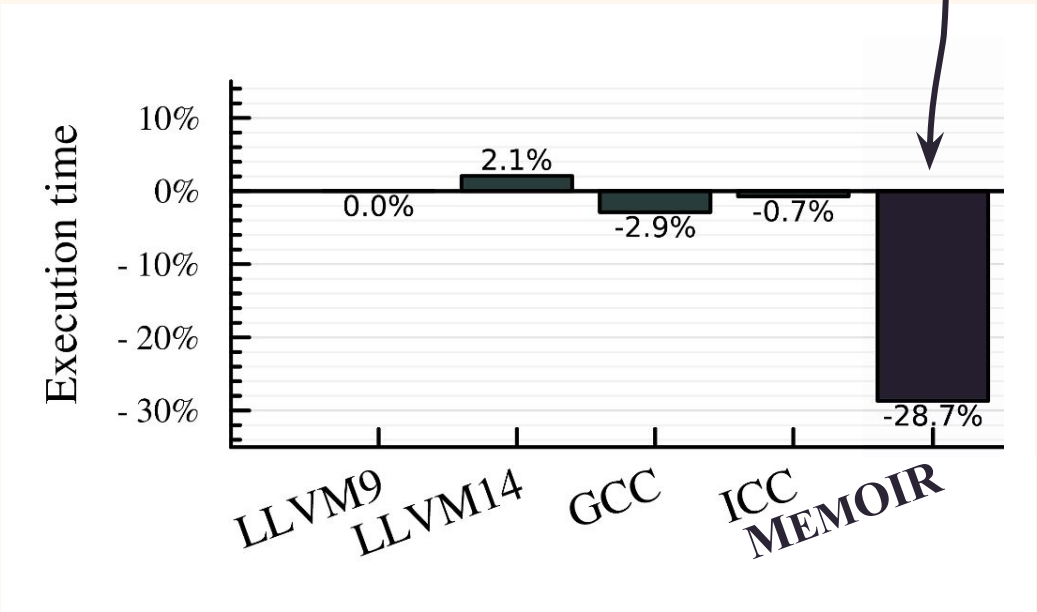


Figure: Execution time of `mcf_s` with `refspeed` input.
10 trials. Normalized to `LLVM9`.

Evaluation

MEMOIR optimizations *reduce memory usage!*

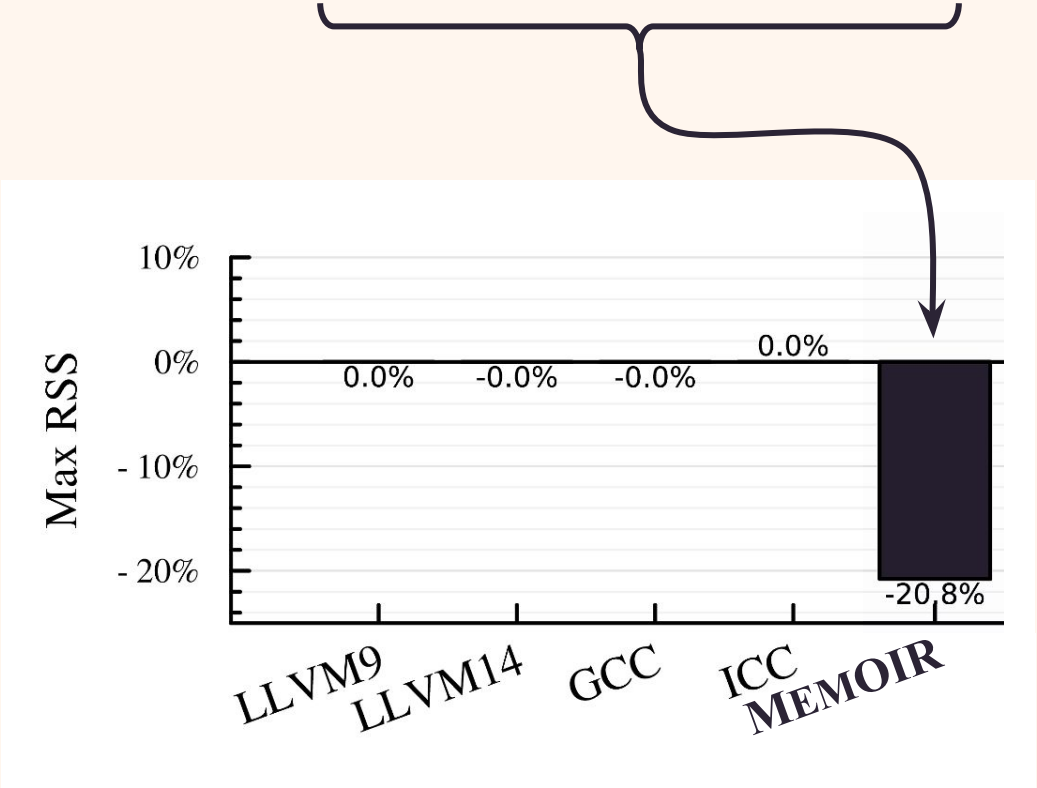


Figure: Execution time of `mcf_s` with `refspeed` input.
10 trials. Normalized to `LLVM9`.

Conclusion

How can I use MEMOIR *today*?

github.com/arcana-lab/memoir

Write a pass with our *open source* compiler

LLVM

NOELLE

Conclusion

How can I use MEMOIR *today*?

github.com/arcana-lab/memoir

Write a pass with our *open source* compiler

- **LLVM**
- **NOELLE**

Write a program using the MEMOIR toolchain

- **C**
- **C++**

Conclusion

How can I use MEMOIR *today*?

github.com/arcana-lab/memoir

Write a pass with our *open source* compiler

- **LLVM**
- **NOELLE**

Write a program using the MEMOIR toolchain

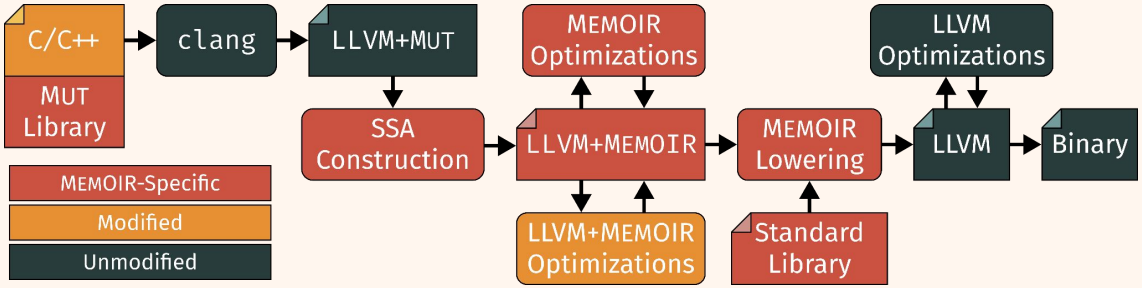
- **C**
- **C++**
- **Rust**
- **Mojo**
- …

More to come!

Representing Data Collections in an SSA Form

Tommy McMichen, Nathan Greiner, Peter Zhong, Federico Sossai, Atmn Patel, Simone Campanoni

github.com/arcana-lab/memoir



www.mcmichen.cc

Northwestern University

ARCANA
Rethinks Compiler Abstractions for New Applications

CONSTELLATION PROJECT

memoir

Most Heap Memory is for Structured Data

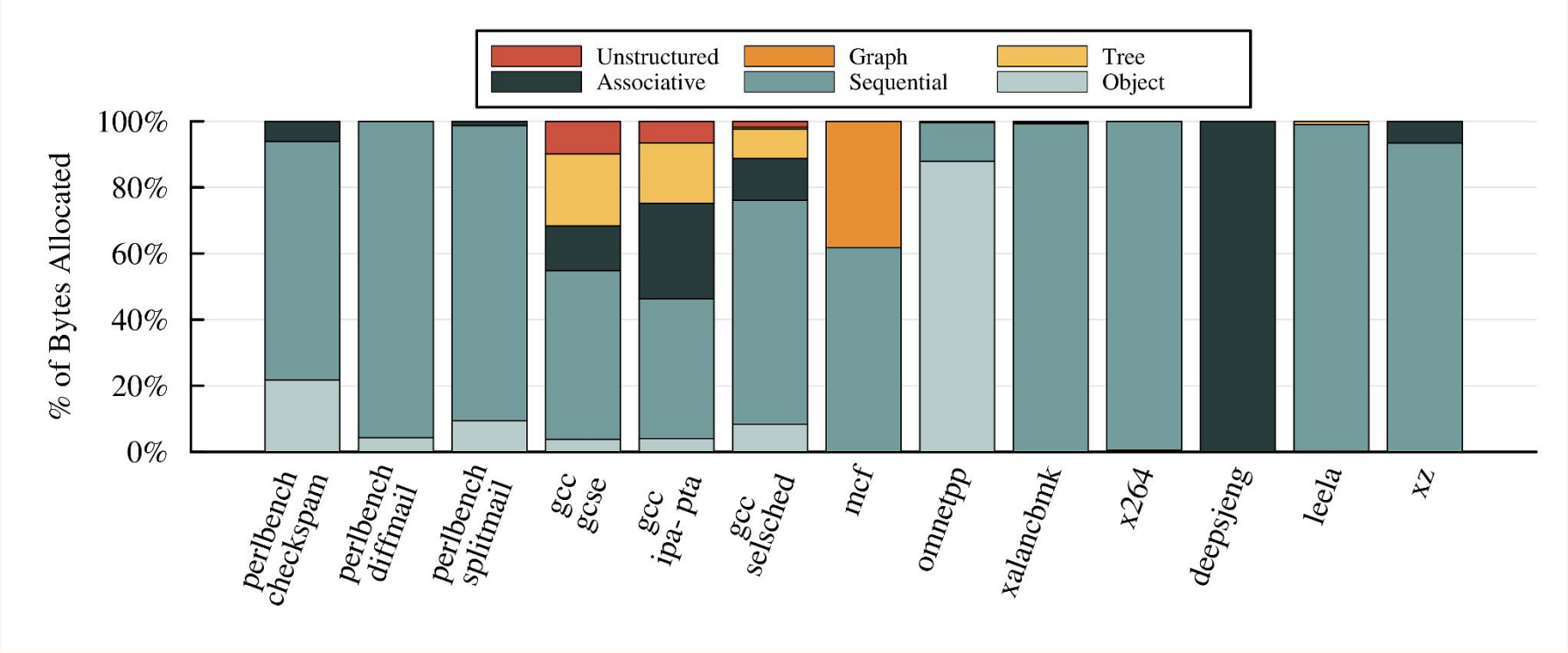


Figure: Breakdown of bytes read and written for each memory class in SPECINT 2017.

Most Reads from Heap Memory are for Structured Data

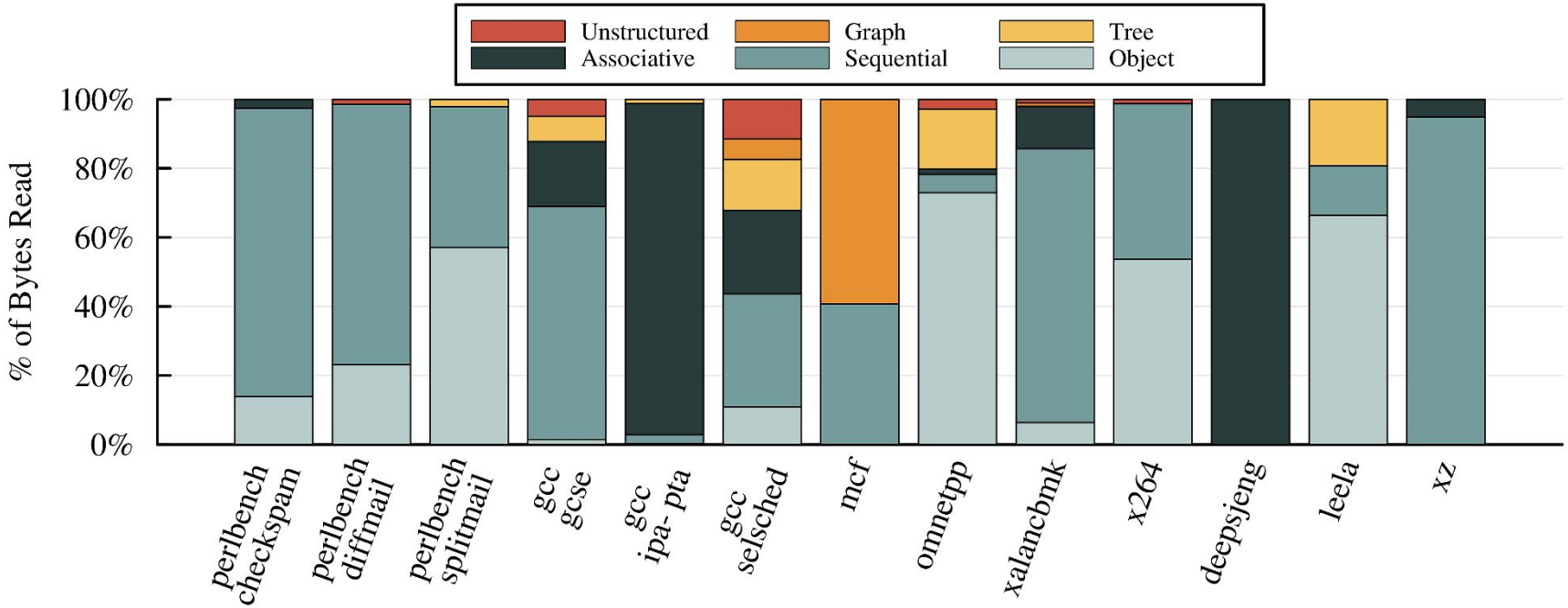


Figure: Breakdown of bytes read and written for each memory class in SPECINT 2017.

Most Writes to Heap Memory are for Structured Data

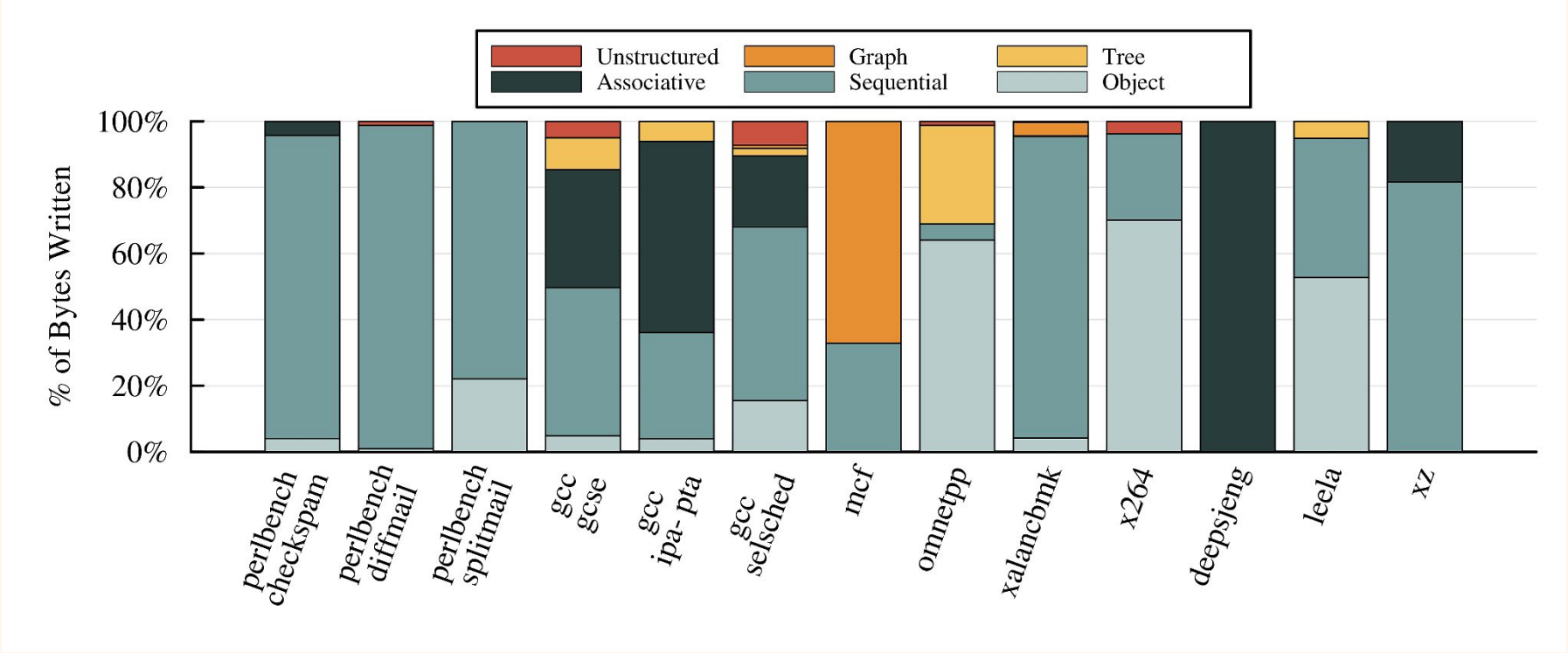


Figure: Breakdown of bytes read and written for each memory class in SPECINT 2017.

Evaluation

MEMOIR requires reasonable compilation time

	Compile Time (ms)			
	MEMOIR		LLVM	
Benchmark	`-O0`	`-O3`	`-O0`	`-O3`
`mcf`	70.6	776.4	20.9	663.2
`deepsjeng`	246.0	1867.6	34.8	852.8
`LLVM opt`	225.9	668.4	52.0	414.7

Motivations

Constant Folding Rarely Succeeds with Memory Operations

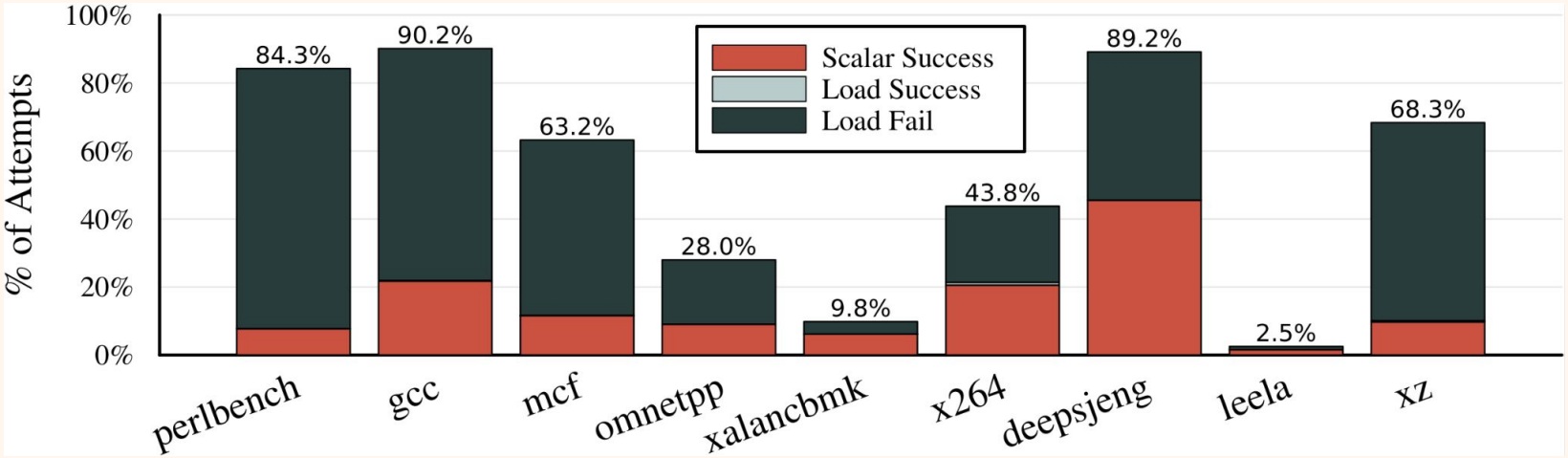


Figure 4: Breakdown of attempts to perform constant folding.

mcf_s Execution Time with Pass Breakdown

Lower is better.

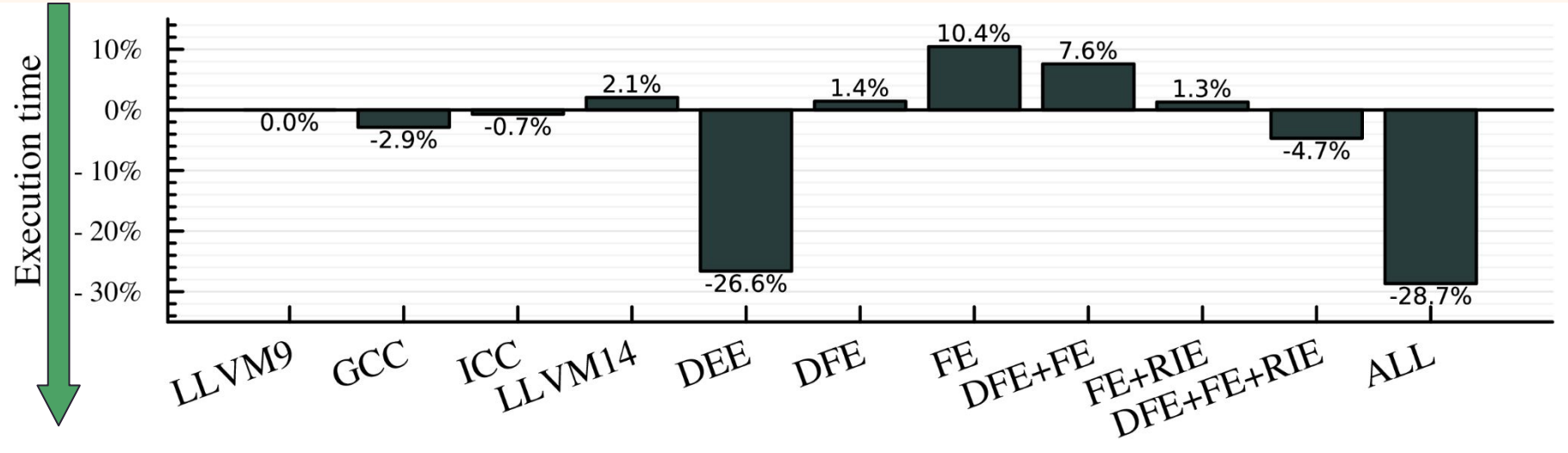


Figure 3: Execution time of `mcf_s` with `refspeed` input.
10 trials. Normalized to `LLVM9`.

mcf_s Max RSS with Pass Breakdown

Lower is better.

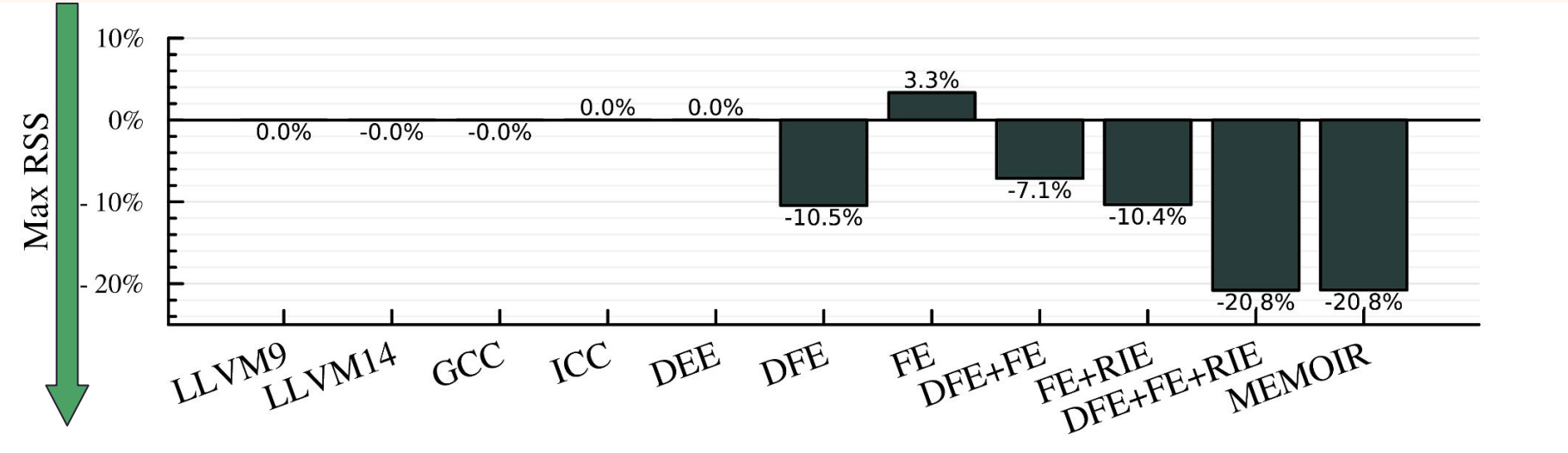
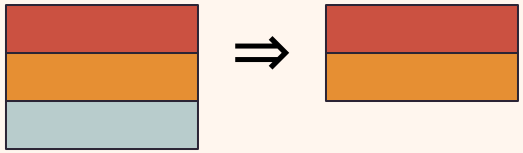


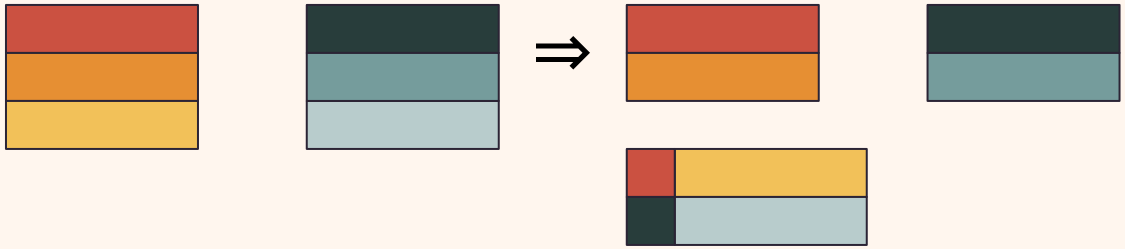
Figure: Maximum resident set size usage of `mcf_s` with `refspeed` input.
10 trials. Normalized to `LLVM9`.

Additional Optimizations

- ***Dead Field Elimination***



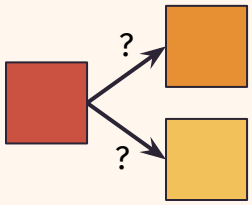
- ***Field Elision***
  - Converts fields of objects into local associative arrays when profitable



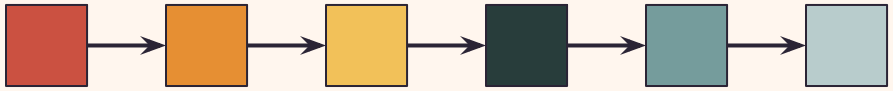
- ***Key Folding***
  - Converts associative arrays into sequences when possible

Linked Data Structures

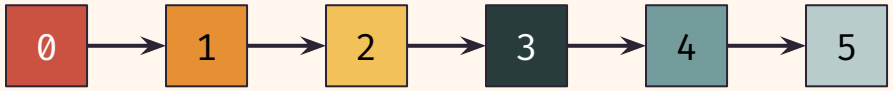
- Linked data structures are the root of undecidability for the memory aliasing problem[1]



- Commonly used for fast insertion and deletion to lists



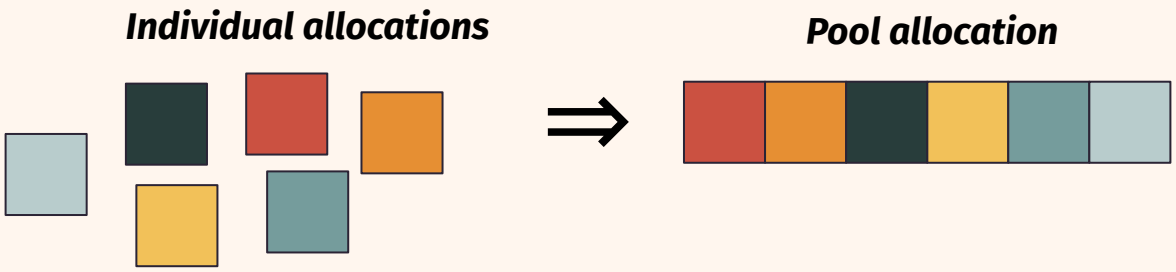
- But this hides the ***underlying index space***, blocking compiler optimizations



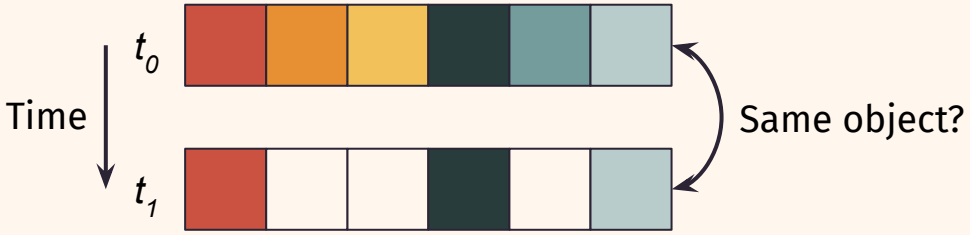
[1]G. Ramalingam. "*The undecidability of aliasing.*" ACM TOPLAS'94, Volume 16, Issue 5.

Premature Memory Optimizations

- Performance engineers employ manual memory optimizations at the source-level to improve memory performance and utilization



- ***Short-lived objects*** are aggregated into the same, ***long-lived object***



- Alias analysis can only glean information about accesses to ***memory locations*** rather than ***memory objects***

Example C to MEMOIR Translation

```
int interp(size_t n, char in[n]) {
  int *stack = malloc(n*sizeof(int))
  size_t sp = 0;
  for (size_t i = 0; i < n; i++) {
    char tok = in[i];
    if (tok == '+') {
       int lhs = stack[sp--];
       int rhs = stack[sp--];
       stack[++sp] = lhs + rhs;
    } else if (tok ⩾ '0'
              && tok ⩽ '9') {
      stack[++sp] = (int)(tok - '0');
    }
  }
  return stack[sp];
}
```

⇒

```
def interp(size n, i8 in[n]) → i32 {
  S0 = new Sequence<i32>(0)
  for (size i = 0; i < n; i++) {
    S1 = ϕ(S0, S5)
    i8 tok = in[i]
    if (tok == '+') {
      i32 lhs = S0[end]
      S1 = USE(S0)
      S2 = S1[0..end-1] // pop
      i32 rhs = S2[end]
      S3 = USE(S2)
      S4 = S3[0..end-1] // pop
      i32 sum = lhs + rhs
      S5 = CONCAT(S4, [sum]) // push
    } else if (tok ⩾ '0' &&
                tok ⩽ '9') {
      i32 val = tok - '0'
      S6 = CONCAT(S1, [val]) // push
    }
    S7 = ϕ(S5, S6)
  }
  i32 top = S7[end]
  return stack[sp];
}
```

Intermediate Representation

Data collections as ***first-class citizens***

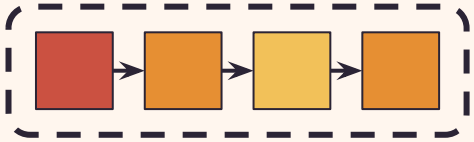
- Collections are ***immutable*** and ***read-once***
  - Their size and elements are fixed upon creation

- New collections can be created by ***adding or removing elements*** from existing collections and ***modifying the elements*** of existing collections

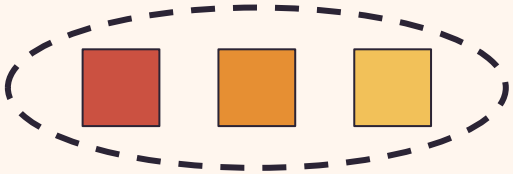
- Collections and their elements have ***static, strong types***

MEMOIR

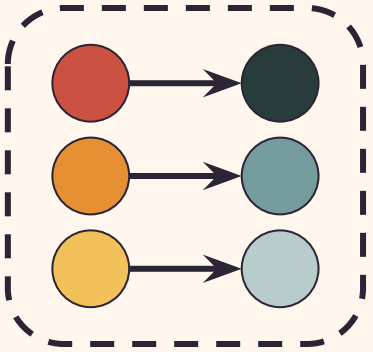
General-Purpose Data Collections



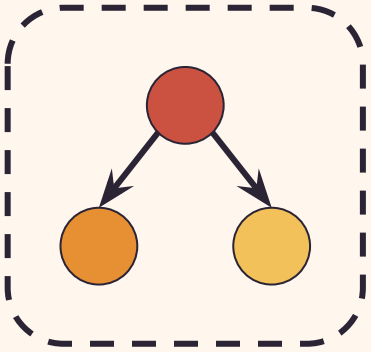
List



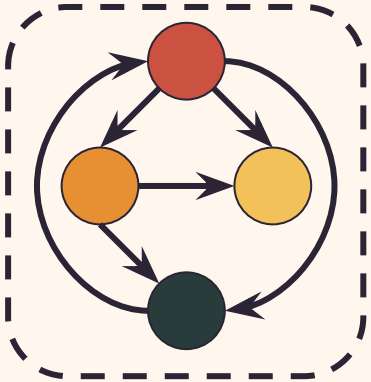
Set



Map



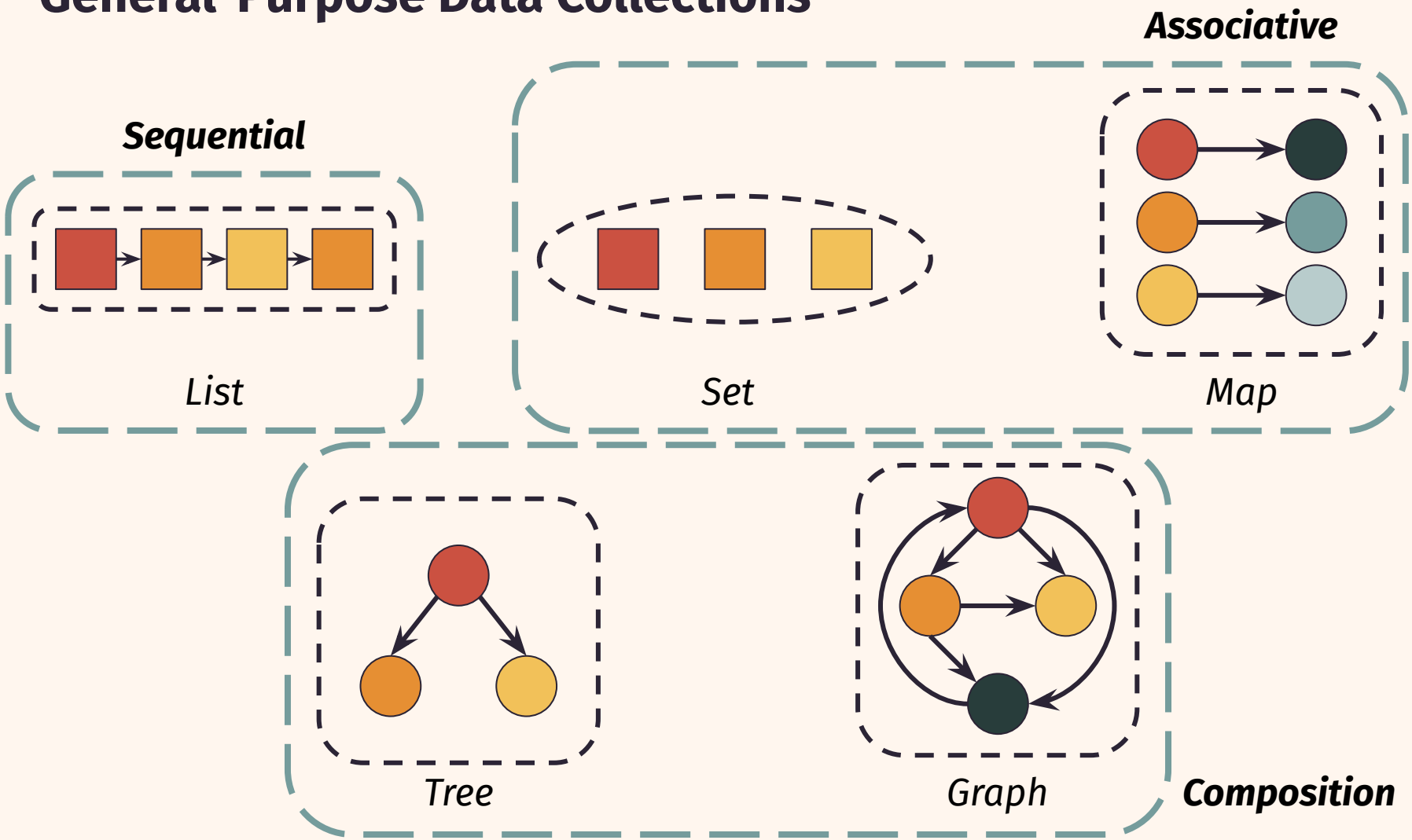
Tree



Graph

MEMOIR

General-Purpose Data Collections



Capture two common cases:

Linked List

Vector

Array

Hashtable

Association List

Bitmap

Capture two common cases:

Linked List

Vector

Array

Sequential

↳ *Contiguous* in index space

Hashtable

Association List

Bitset

Associative

↳ *Distinctness* in index space

Read-once Form

- A read-once form is constructed by ***creating a new collection variable after each read***

$$\ldots = S_0[\%i]$$

$$S_1 = \textbf{USE}\phi(S_0[\%i], S_0)$$

$$\ldots = A_0[\%k]$$

$$A_1 = \textbf{USE}\phi(A_0[\%k], A_0)$$

Data Flow of Collections

- Collections may only be accessed via DEF-USE data flow
  - ***Cannot escape*** into memory locations

Object Types

- User-defined product types are supported

```
struct T {          ⇒   type T = {
  int x;                  x: i32,
  float y;                y: f32,
  int *z;                 z: ptr,
  T *n;                   n: &T
};                      }
```

- Support for primitive types
  - Raw pointers are handled by the `ptr` type
- Support for reference types
  - Nullable reference to a MEMOIR object of the given type

- Additional constraints for field ordering and alignment can be specified

Objects

- Objects are an instantiation of a given object type

```
@0 = new T
```

- Fields are laid out according to the type
  - If packing or padding are needed, it must be explicit in the type definition

Field Arrays

- Fields of an object are accessed via ***field arrays***: an associative array from an object reference to the field's value

```
F^T.x = new Assoc<&T,i32>
F^T.y = new Assoc<&T,f32>
F^T.z = new Assoc<&T,ptr>
F^T.n = new Assoc<&T,&T>
```

- ***Decouples the memory layout of fields within an object from their access***

Live-Variable Analysis

- Determine if a ***variable*** is alive at a given program point

```
a = 1                ▷ { a }
b = a + 1            ▷ { a, b }
c = a * 2            ▷ { b, c }
d = b + c            ▷ { d }
```

Live-Element Analysis

- Determine if an ***element*** is alive at a given program point

```
a[0] = 1                  ▷ { a[0] }
a[1] = a[0] + 1       ▷ { a[0], a[1] }
a[2] = a[0] * 2       ▷ { a[1], a[2] }
a[3] = a[1] + a[2]        ▷ { a[3] }
```

Live-Element Analysis

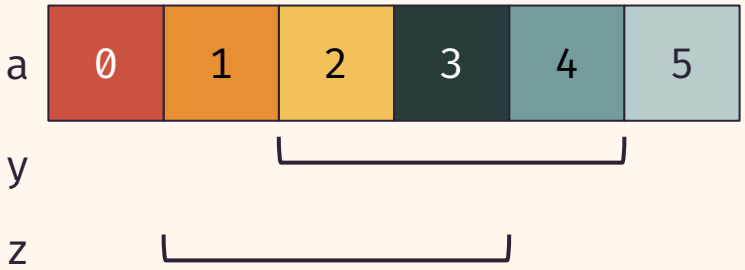
- Unlike scalars, the ***element index*** is not necessarily known at compile time:

```
a[i] = 1                  ▷ { a[i] }
a[j] = a[i] + 1           ▷ { a[i], a[j] }
a[k] = a[i] * 2           ▷ { a[i], a[j], a[k] }
a[x] = a[y] + a[z]        ▷ { a[i], a[j], a[k], a[x] }
```

Live-Range Analysis

↳ Value range analysis informs us of elements that ***may be used***:

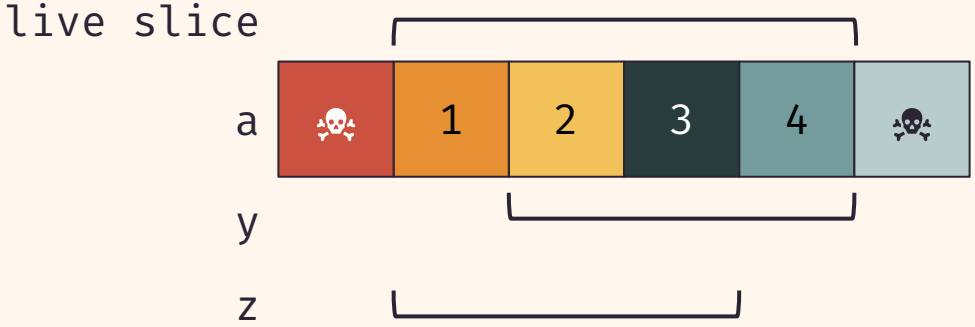
```
a[i] = 1                    ▷ { a[i] }
a[j] = a[i] + 1             ▷ { a[i], a[j] }
a[k] = a[i] * 2             ▷ { a[i], a[j], a[k] }
a[x] = a[y] + a[z]          ▷ { a[i], a[j], a[k], a[x] }
```



Live-Range Analysis

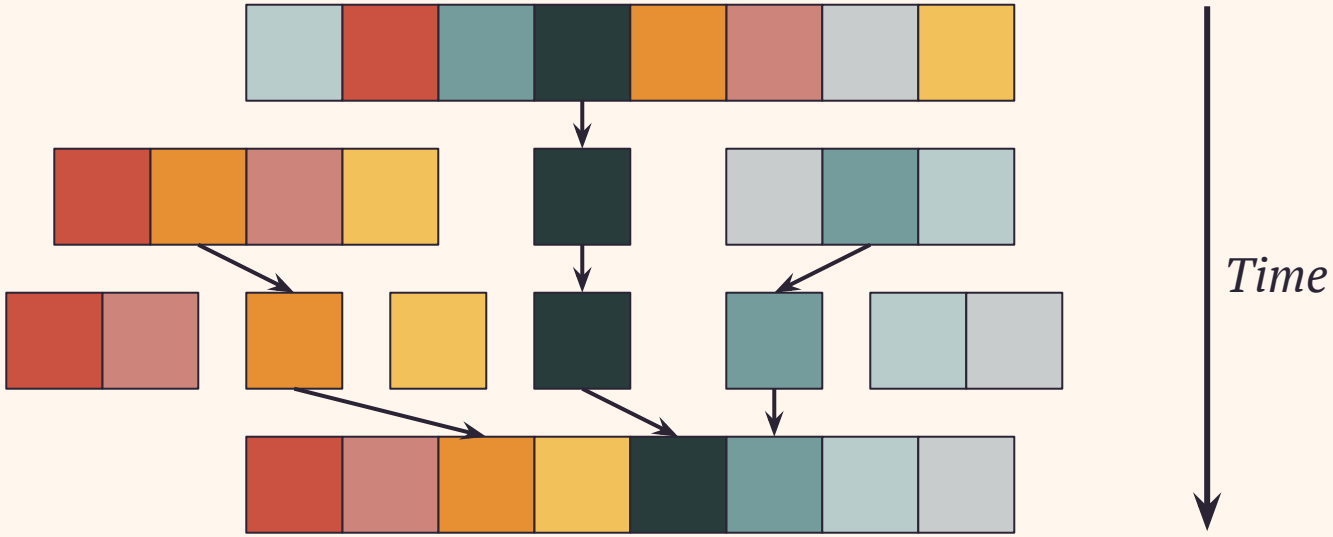
↳ So we can determine ***live ranges*** of collection index spaces:

```
a[i] = 1                    ▷ { a[i] }
a[j] = a[i] + 1             ▷ { a[i], a[j] }
a[k] = a[i] * 2             ▷ { a[i], a[j], a[k] }
a[x] = a[y] + a[z]          ▷ { a[i], a[j], a[k], a[x] }
```

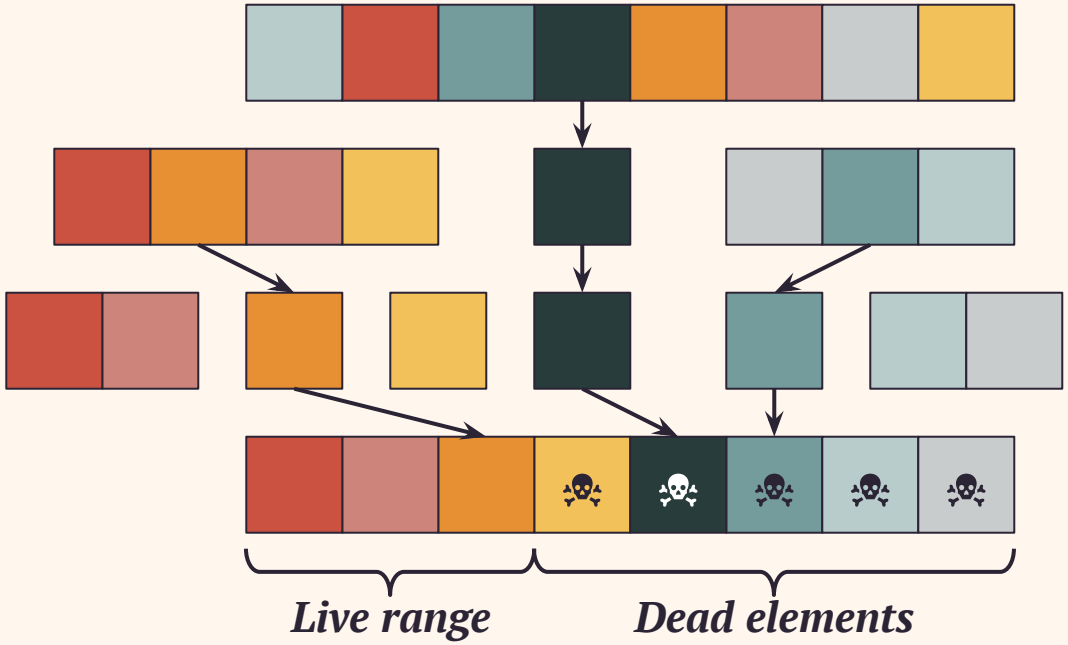


Example Application: mcf_s from SPEC2017

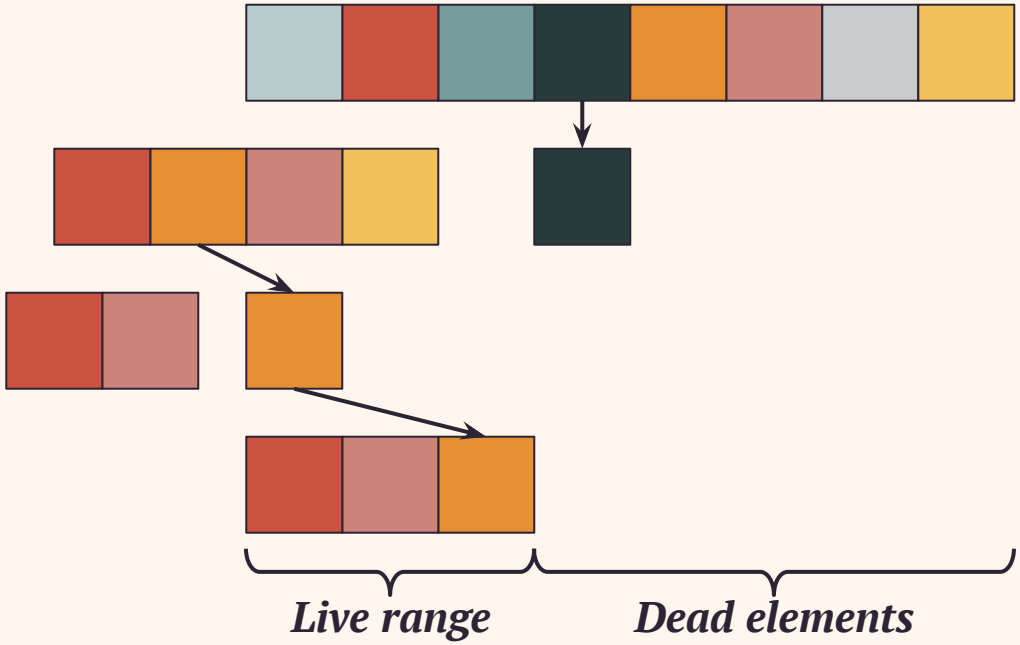
Quick sort accounts for ***39.9%*** of exec. time



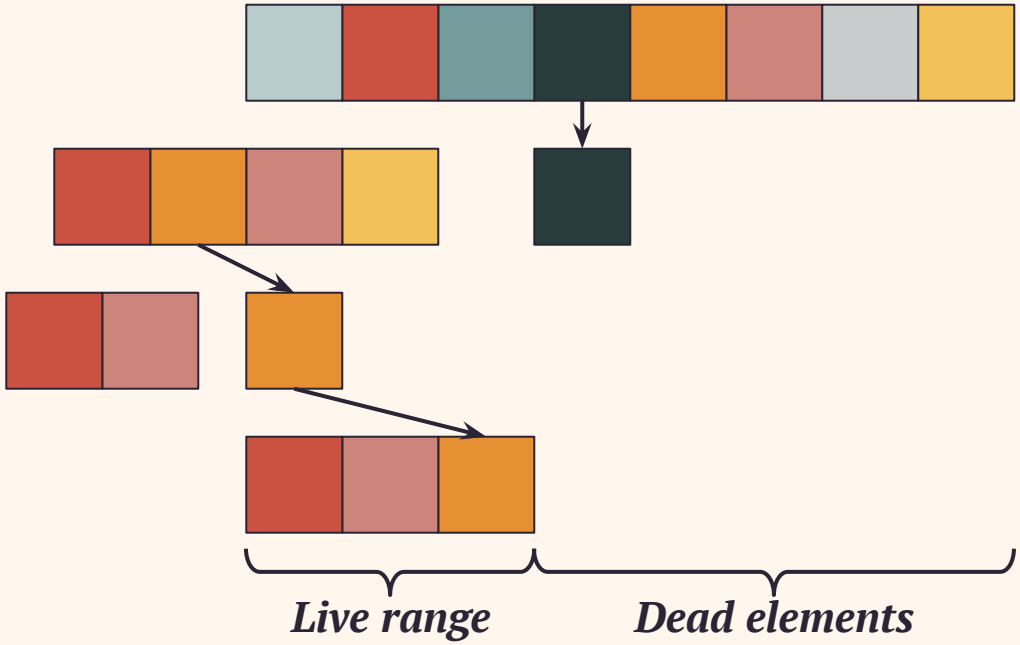
Live-Range Analysis reports only a portion of the result is needed.



Dead Element Elimination converts Sort → Partial Sort



Dead Element Elimination converts Sort → Partial Sort



With no primitive knowledge of sort!

mcf_s parallel speedup with DEE optimization