Testing the Limits of Allocator Awareness

Bob Steagall

C++Now 2017

Lower Testing the Limits of Allocator Awareness

Bob Steagall

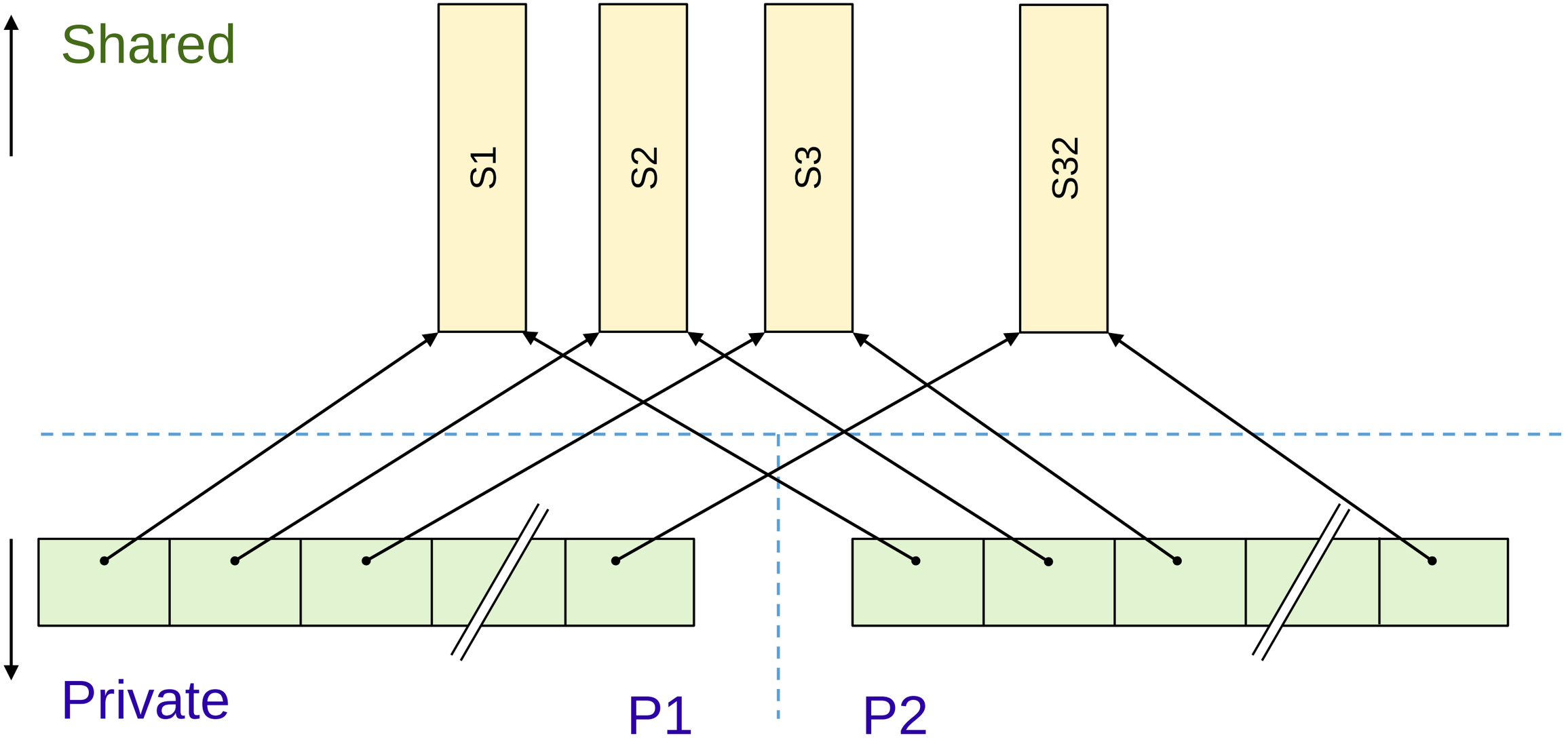
C++Now 2017

Overview

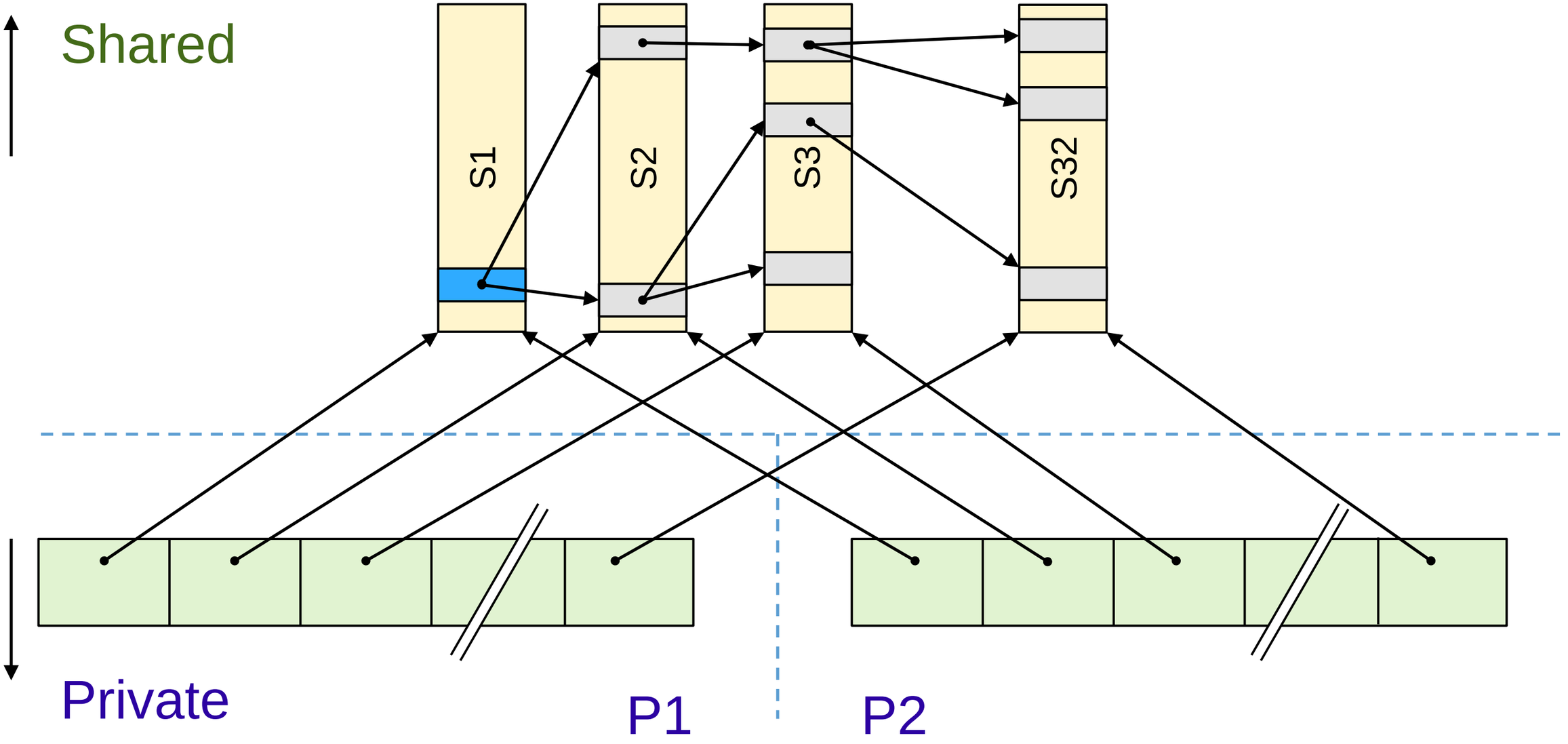
- Motivation
- Some allocator background
- Building a test suite
- Synthetic pointer performance testing
- Allocator awareness conformance testing
- Summary

Motivation

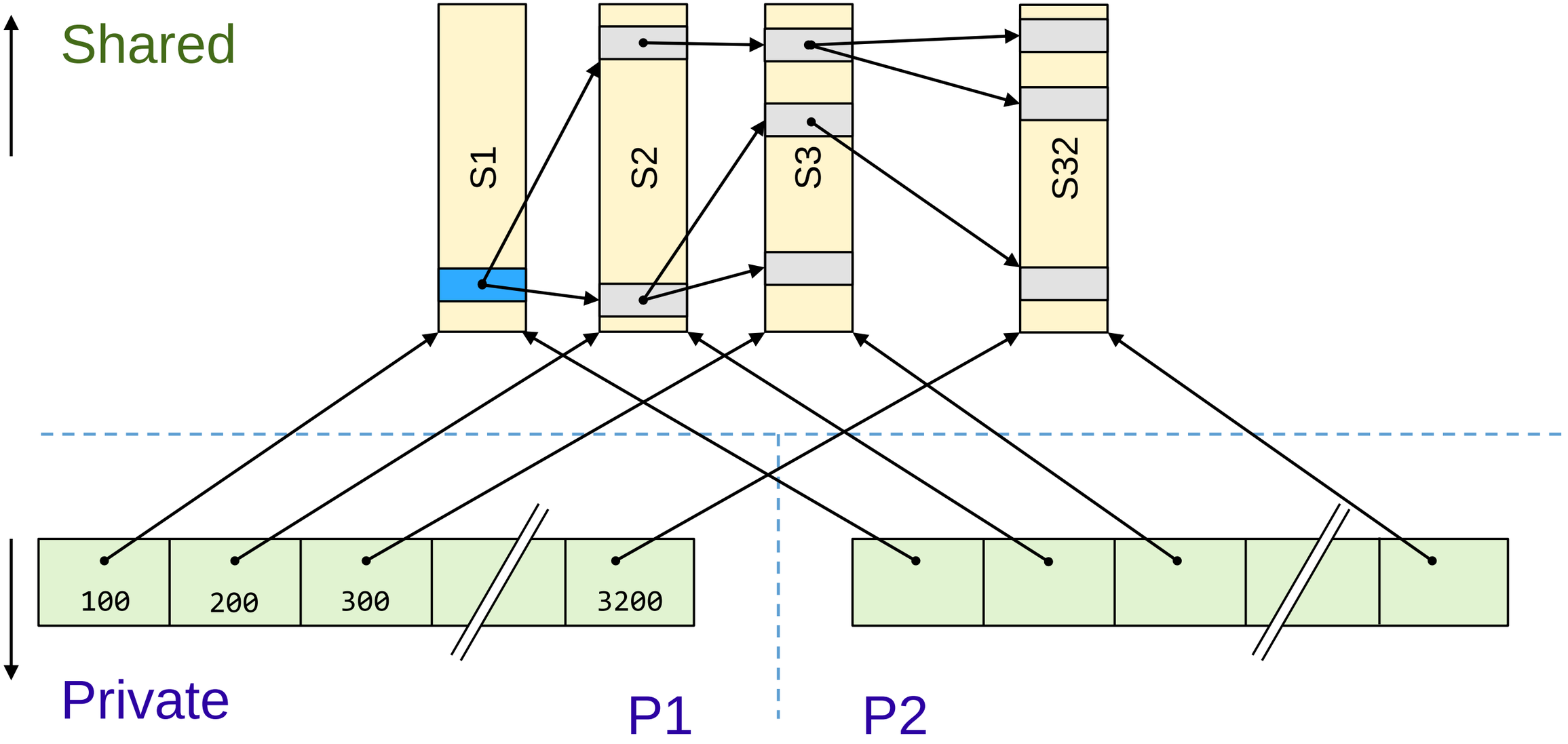
Motivating Example – Shared Memory Database



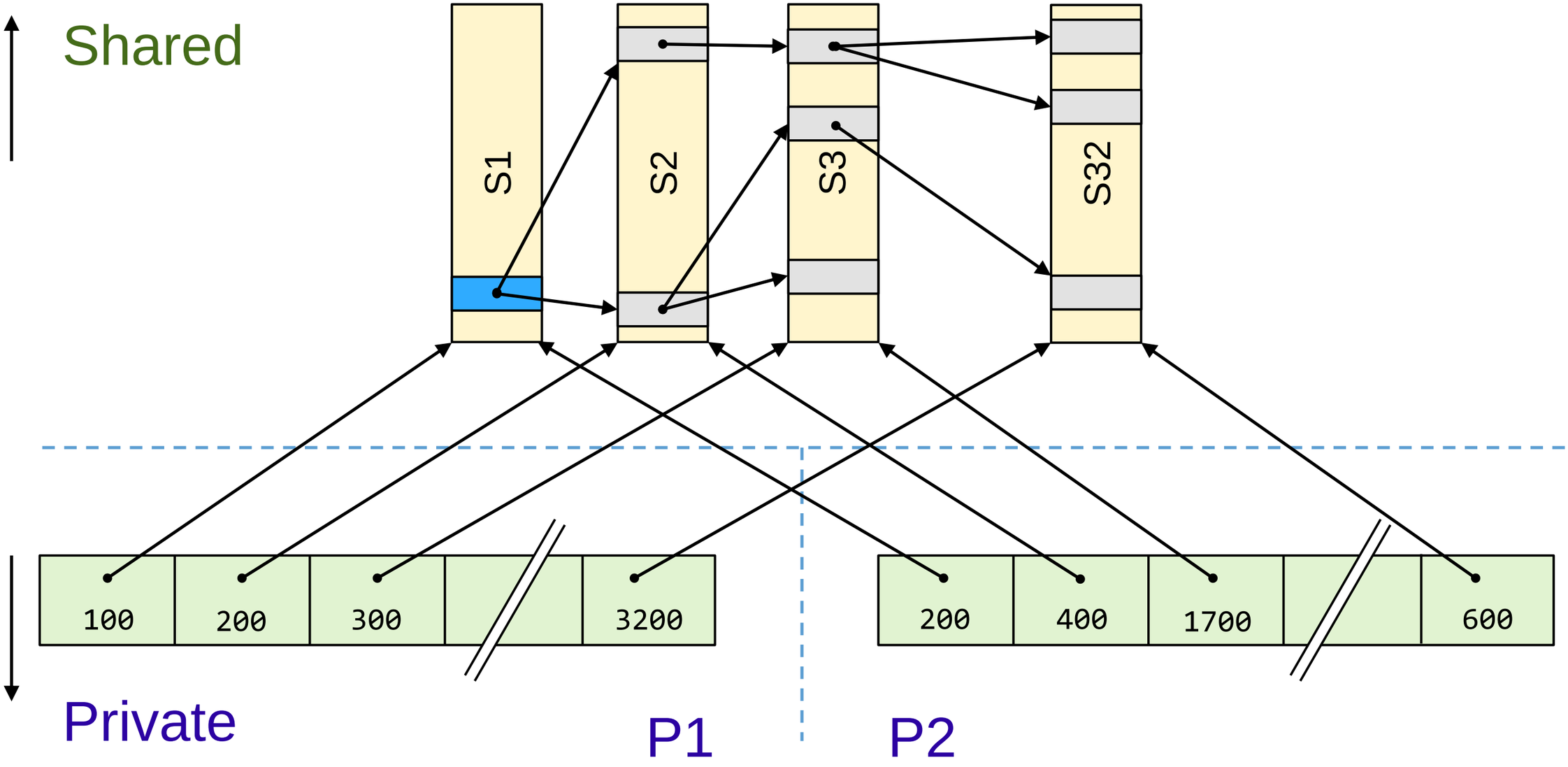
Motivating Example – Shared Memory Database



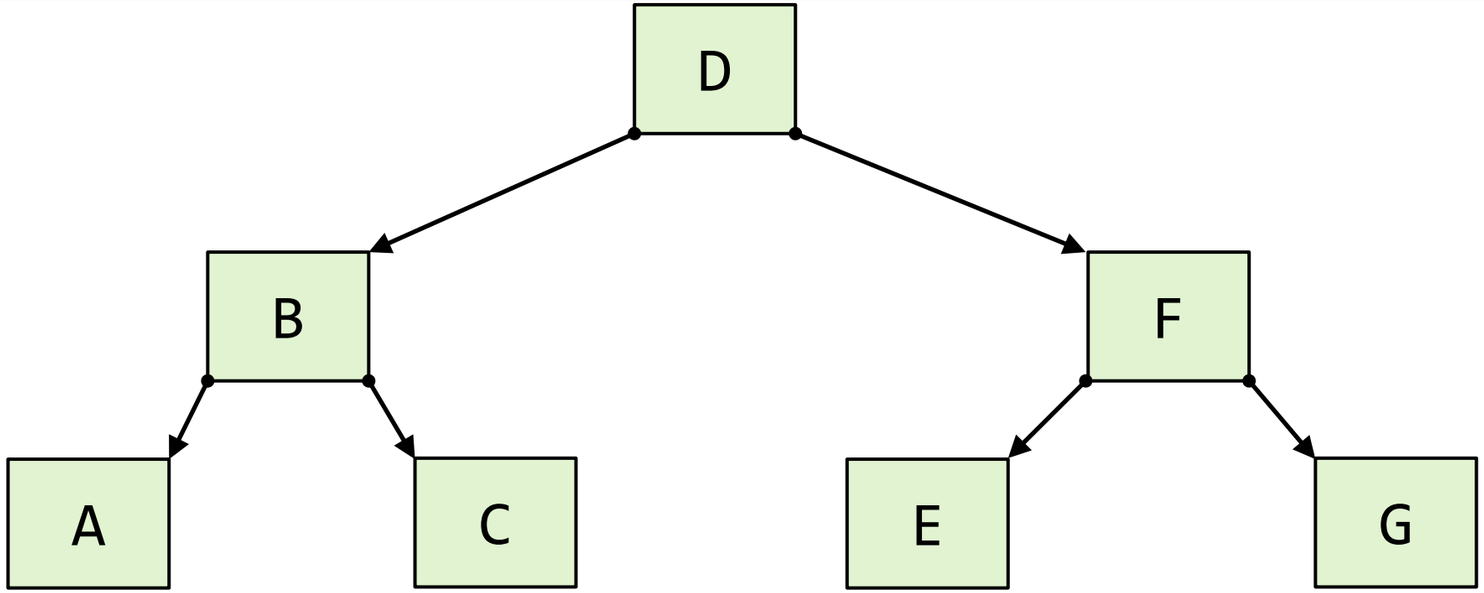
Motivating Example – Shared Memory Database



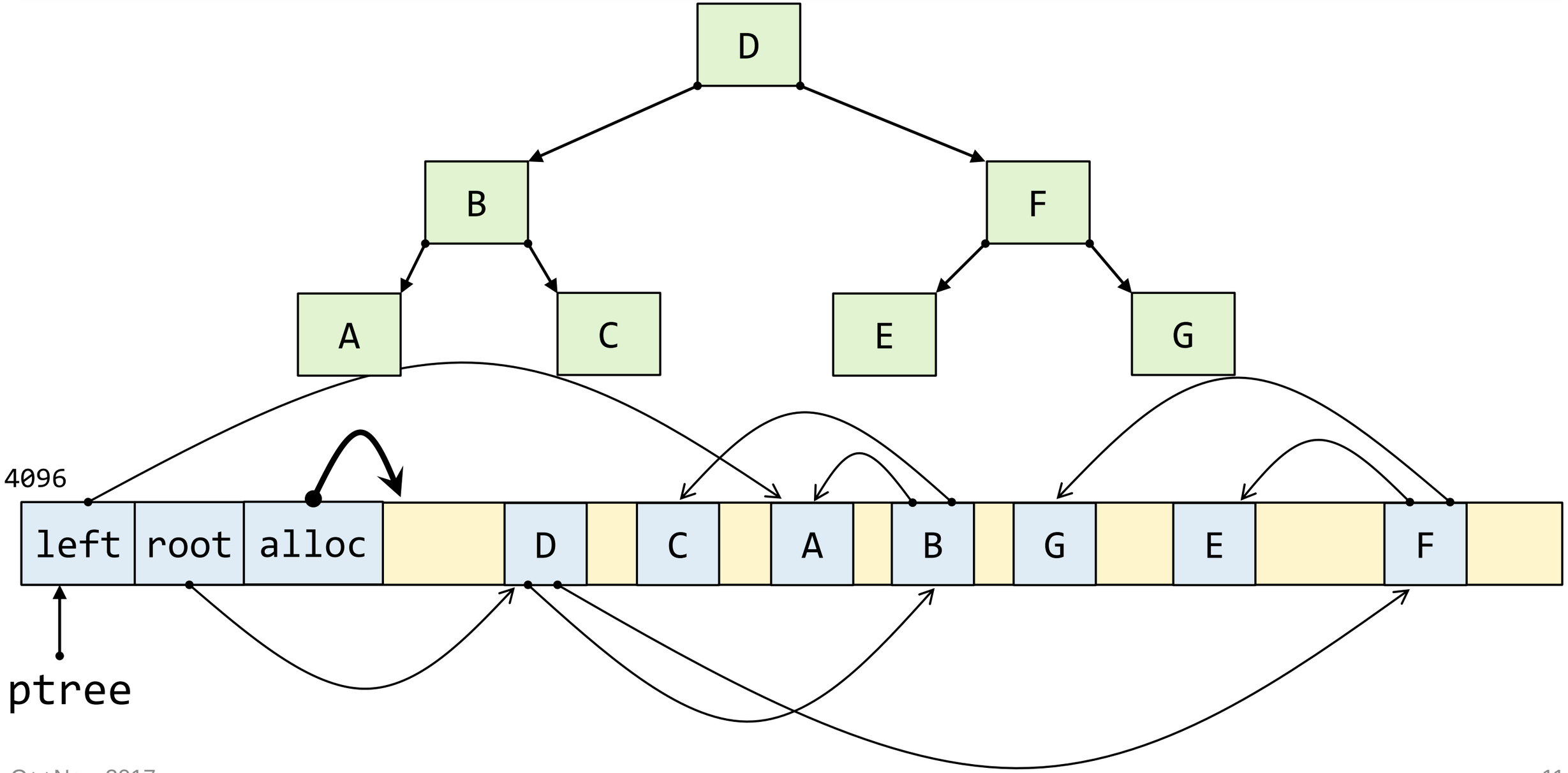
Motivating Example – Shared Memory Database



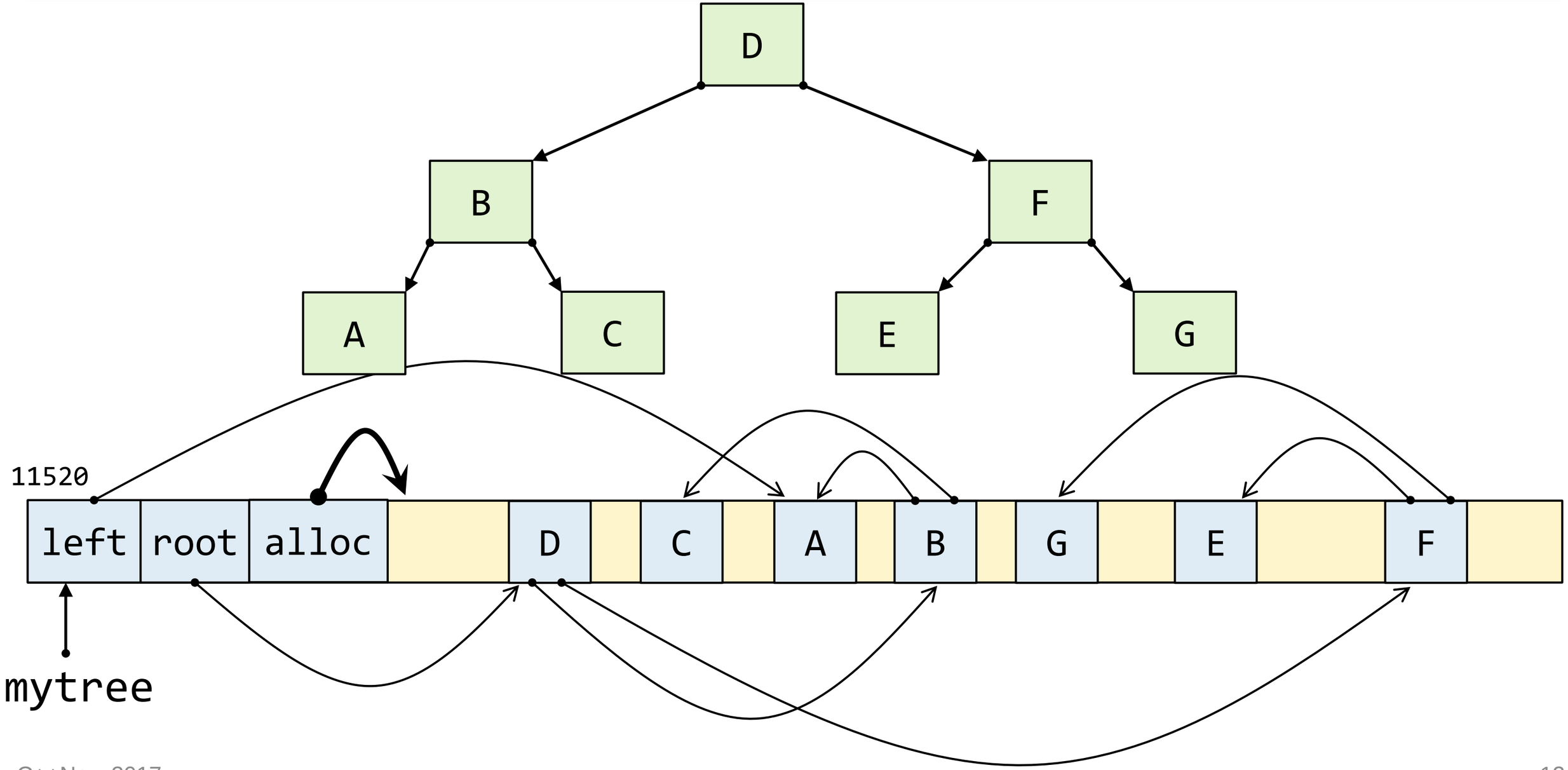
Motivating Example – Self-Contained DOM



Motivating Example – Self-Contained DOM



Motivating Example – Self-Contained DOM



Some Background

The Brilliance of Allocators

- Allocators abstract memory management details away from containers
- Encapsulate information about addressing (memory model)
- Allocators separate allocation/deallocation from construction/destruction

- Containers don't need to understand allocation strategies or addressing
- Containers obtain allocator services through parameterization
- Containers can work with chunks of raw bytes, and construct/destroy objects when it makes the most sense

Allocators Before C++11

```
template<class T>
struct allocator
{
    typedef size_t     size_type;
    typedef ptfdiff_t  difference_type;
    typedef T*         pointer;
    typedef T const*   const_pointer;
    typedef T&         reference;
    typedef T const&   const_reference;
    typedef T          value_type;

    template<class U>  struct rebind { typedef allocator<U> other; };

    ...

    pointer allocate(size_type, allocator<void>::const_pointer hint = 0);
    void    deallocate(pointer p, size_type n);

    void    construct(pointer p, T const& val);
    void    destroy(pointer p);
};
```

Allocators Before C++11

- 14882:2003 / 20.1.5.4

  Implementations of containers described in this International Standard are permitted to assume that their Allocator template parameter meets the following two additional requirements beyond those in Table 32.

  —All instances of a given allocator type are required to be interchangeable and always compare equal to each other.

  — The typedef members pointer, const_pointer, size_type, and difference_type are required to be T*, T const*, size_t, and ptrdiff_t, respectively.

- 14882:2003 / 20.1.5.5

  Implementors are encouraged to supply libraries that can accept allocators that encapsulate more general memory models and that support non-equal instances. In such implementations, any requirements imposed on allocators by containers beyond those requirements that appear in Table 32, and the semantics of containers and algorithms when allocator instances compare non-equal, are implementation-defined.

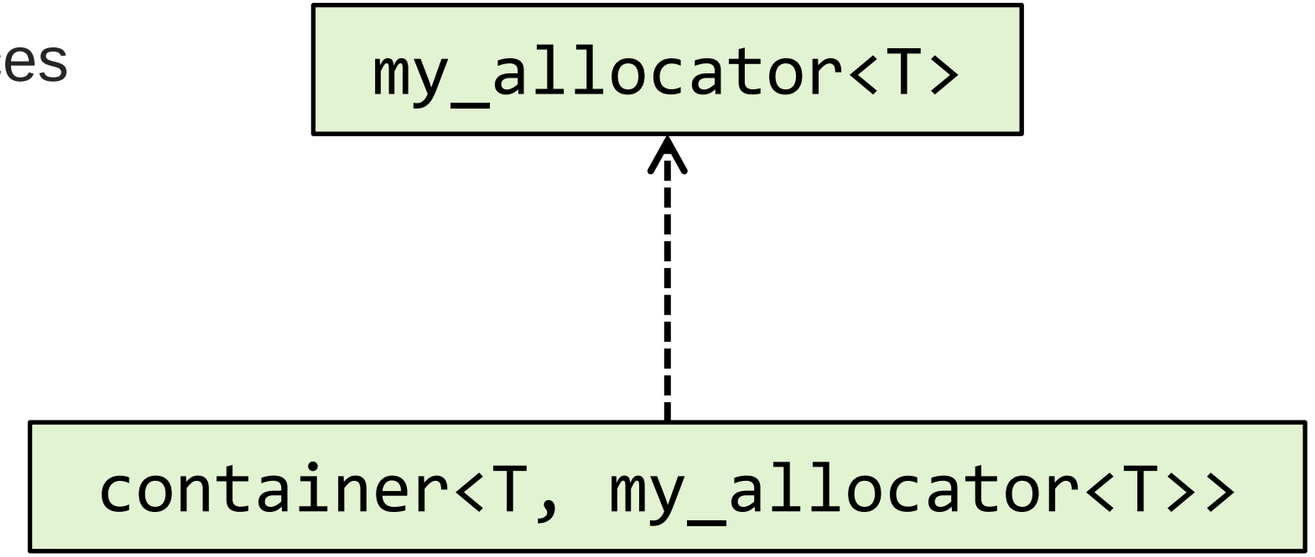
Allocators Before C++11

- Containers obtain their allocation services from the allocator template argument:

```
void* p = my_alloc.allocate(n);
```

- May assume that:

```
using pointer = T*;
using const_pointer = T const*;
using void_pointer = void*;
using const_void_pointer = void const*;
using size_type = size_t;
using difference_type = ptrdiff_t;
my_allocator<Foo>  a, b;  ➔ a == b;
```



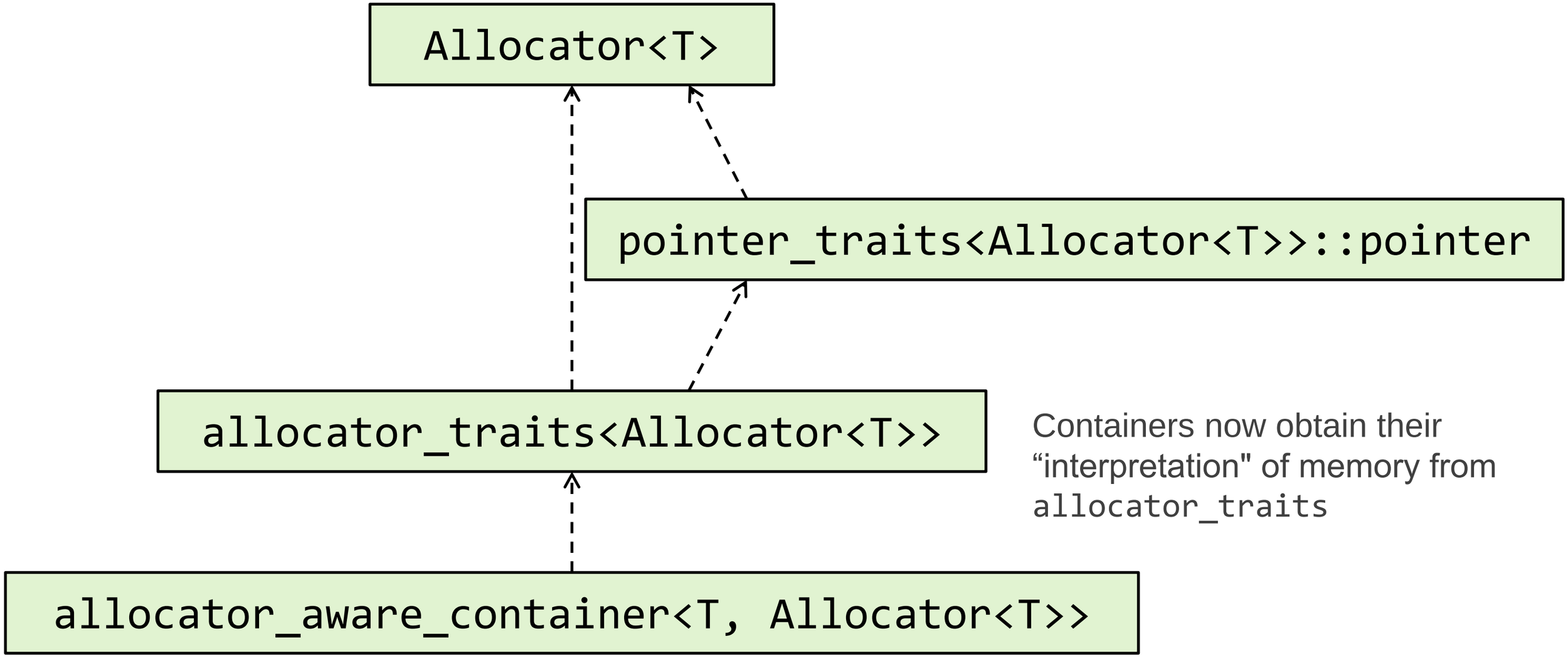
Allocators Before C++11 – Implications

- Standard containers did not have to support synthetic pointers
- Standard containers did not have to support stateful allocators
- Large class of interesting problems could not be solved using standard containers

Allocators After C++11

- Paragraphs 20.1.5.4 / 5 deleted!
- New requirements to improve allocators
  - *nullablepointer.requirements*
    - Pointer-like type that supports null values
  - *allocator.requirements*
    - Defines allocator and relationship to allocator traits
  - *pointer.traits*
    - Describes a uniform interface to pointer-like types
  - *allocator.traits*
    - Describes uniform interface to allocator types
  - *allocator.adaptor*
    - Describes adaptor that supports deep propagation of allocators.
  - *container.requirements.general*
    - Defines allocator-aware container requirements

Allocators After C++11



Containers now obtain their “interpretation" of memory from `allocator_traits`

Allocator Traits

```
template <class Alloc>
struct allocator_traits
{
    typedef typename Alloc::value_type value_type;
    typedef Alloc   allocator_type;
    typedef CHOOSE  pointer;
    typedef CHOOSE  const_pointer;
    typedef CHOOSE  void_pointer;
    typedef CHOOSE  const_void_pointer;
    typedef CHOOSE  difference_type;
    typedef CHOOSE  size_type;

    typedef CHOOSE  propagate_on_container_copy_assignment;   //- POCCA
    typedef CHOOSE  propagate_on_container_move_assignment;   //- POCMA
    typedef CHOOSE  propagate_on_container_swap;              //- POCS
    typedef CHOOSE  is_always_equal;

    template <class T> using rebind_alloc  = CHOOSE ;
    template <class T> using rebind_traits = allocator_traits<rebind_alloc<T>>;
    ...
};
```

Allocator Traits

```
template<class Alloc>
struct allocator_traits
{
    ...

    static  pointer allocate(Alloc& a, size_type n);
    static  pointer allocate(Alloc& a, size_type n, const_void_pointer hint);
    static  void    deallocate(Alloc& a, pointer p, size_type n);

    template <class T, class... Args>
    static  void    construct(Alloc& a, T* p, Args&&... args);
    template <class T>
    static  void    destroy(Alloc& a, T* p);

    static Alloc    select_on_container_copy_construction(const Alloc& rhs);

    ...
};
```

What are the Implications of Allocator Awareness?

- For `std` library users – not much
- Allocator implementors have to consider question of allocator propagation in 5 cases
  - copy/move construction
  - copy/move assignment
  - swap
- Library implementors had to adapt the `std` containers accordingly in the same 5 cases
  - How are allocator members constructed?
  - When are allocator members replaced?

Example Container

```
template<class T, class Alloc<T>>
class my_container
{
  public:
    typedef T                                       value_type;
    typedef Allocator                               allocator_type;
    typedef std::allocator_traits<allocator_type>   alloc_traits;

    typedef typename alloc_traits::size_type        size_type;
    typedef typename alloc_traits::difference_type  difference_type;
    typedef typename alloc_traits::pointer          pointer;
    typedef typename alloc_traits::const_pointer    const_pointer;
    typedef typename alloc_traits::reference        reference;
    typedef typename alloc_traits::const_reference  const_reference;

    typedef stuff                                   iterator;
    typedef const_stuff                             const_iterator;
    typedef std::reverse_iterator<iterator>         reverse_iterator;
    typedef std::reverse_iterator<const_iterator>   const_reverse_iterator;
    ...
};
```

Example Container

```
template<class T, class Alloc<T>>
class my_container
{
  public:
    ...
    my_container(my_container const& other);
    my_container(my_container const& other, allocator_type const& alloc);
    my_container(my_container&& other);
    my_container(my_container&& other, allocator_type const& alloc);

    my_container&  operator =(my_container const& other);
    my_container&  operator =(my_container&& other);

    ...

    swap(my_container& other);

  private:
    rep_types       m_rep;
    allocator_type  m_alloc;
};
```

Copy Construction

```
my_container::my_container(my_container const& other)
:   m_rep()
,   m_alloc(traits_type::select_on_container_copy_construction(other.m_alloc))
{
     this->assign_from(other.cbegin(), other.cend());
}


my_container::my_container(my_container const& other, allocator_type const& alloc)
:   m_rep()
,   m_alloc(alloc)
{
    this->assign_from(other.cbegin(), other.cend());
}
```

Move Construction

```
my_container::my_container(my_container&& other)
:   m_rep()
,   m_alloc(std::move(other.m_alloc))
{
    this->move_rep_from(other);
}


my_container::my_container(my_container&& other, allocator_type const& alloc)
:   m_rep()
,   m_alloc(alloc)
{
    this->move_rep_from(other);
}
```

Copy Assignment

```
my_container&
my_container::operator =(my_container const& other)
{
    if (&other != this)
    {
        if (alloc_traits::POCCA == std::true_type)
        {
            if (this->m_alloc != other.m_alloc)
            {
                this->clear_and_deallocate_memory();
            }
            this->m_alloc = other.m_alloc;
        }
        this->assign_from(other.cbegin(), other.cend());
    }
}
```

Move Assignment

```
my_container&
my_container::operator =(my_container&& other)
{
    if (alloc_traits::POCMA == std::true_type)
    {
        this->clear_and_deallocate_memory();
        this->move_alloc_from(other);
        this->move_rep_from(other);
    }
    else if (this->m_alloc == other.m_alloc)
    {
        this->clear_and_deallocate_memory();
        this->move_rep_from(other);
    }
    else
    {
        this->assign_from(std::make_move_iterator(other.begin()),
                          std::make_move_iterator(other.end()));
    }
}
```

Swap

```
my_container&
my_container::swap(my_container& other)
{
    if (&other != this)
    {
        if (alloc_traits::POCS == std::true_type)
        {
            std::swap(this->m_rep, other.m_rep);
            std::swap(this->m_alloc, other.m_alloc);
        }
        else if (this->m_alloc == other.m_alloc)
        {
            std::swap(this->m_rep, other.m_rep);
        }
        else
        {
            //- Undefined behavior.
        }
    }
}
```

Building a Test Suite

Building a Test Suite

- What to test?
  - The space of testable things is vast
  - Think in two dimensions

pointer \ allocator	stateless	stateful
ordinary	`std::allocator<T>`	?
synthetic	?	?

Building a Test Suite

- Confine efforts to
  - Synthetic pointer performance tests
  - Basic container conformance tests

- Confine scope to
  - A small number of algorithms for pointer performance tests
    - `copy()`, `sort()`, `stable_sort()`
  - Subset of containers and strings for container conformance tests
    - `deque`, `forward_list`, `list`, `map`, `string*`, `unordered_map`, `vector`

Building a Test Suite

- How to define success?
  - Compilation and linking succeeds
  - Tests run without crashing, hangs, or infinite loops
  - Synthetic pointer performance tests yield same results as ordinary pointers
  - Container conformance tests using test allocators of `<T>` yield same runtime results as `std::allocator<T>`
  - Containers support relocation (relocatable memory segments)**
- How to build the tests?
  - Implement certain aspects of allocator design as policy types
  - Re-use some prior work

Thinking (Slightly) Differently About Memory Allocation

- Structural Management
  - Addressing Model
  - Storage Model
  - Pointer Interface
  - Allocation Strategy
- Concurrency Management
  - Thread Safety
  - Transaction Safety

Concept – Addressing Model

- Policy type that implements primitive addressing operations
  - Analogous to `void*`
  - Convertible to `void*`
- Internally, the addressing model defines
  - The bits used to represent an address
  - How an address is computed
  - How memory is arranged
- Representations
  - Ordinary pointer `void*` (aka natural pointer)
  - Synthetic `void` pointer (aka fancy pointer, pointer-like type)

Concept – Storage Model

- Policy type that manages segments
  - Interacts with an external source of memory to borrow and return segments
    - Segment: a region of memory that has been provided to the storage model by some external source
  - Provides an interface to segments in terms of the addressing model
  - Lowest-level allocation
- Some sources of segments
  - `brk()` / `sbrk()` / `mmap()` — Unix private memory
  - `VirtualAlloc()` / `HeapAlloc()` — Windows private memory
  - `shmget()` / `shmat()` — System V shared memory
  - `CreateFileMapping()` / `MapViewOfFile()` — Windows shared memory

Concept – Pointer Interface

- Policy type that wraps the addressing model to emulate a pointer to data
  - Analogous to T*
  - Provides (enough) pointer syntax
  - Is convertible "in the right direction" to ordinary pointers
  - Is convertible "in the right direction" to other pointer interface types

- Representations
  - Ordinary pointer T* (aka, natural pointer)
  - Synthetic pointer (aka fancy pointer, pointer-like type)

Concept - Allocation Strategy

- Policy type that manages the process of allocating memory for clients
  - Requests segment allocation/deallocation from the storage model
  - Interacts with segments in terms of the addressing model
  - Divides segments into chunks
    - Chunk: A region of memory carved out of a segment to be used by an allocator's client
  - Provides chunks to the client in terms of the pointer interface

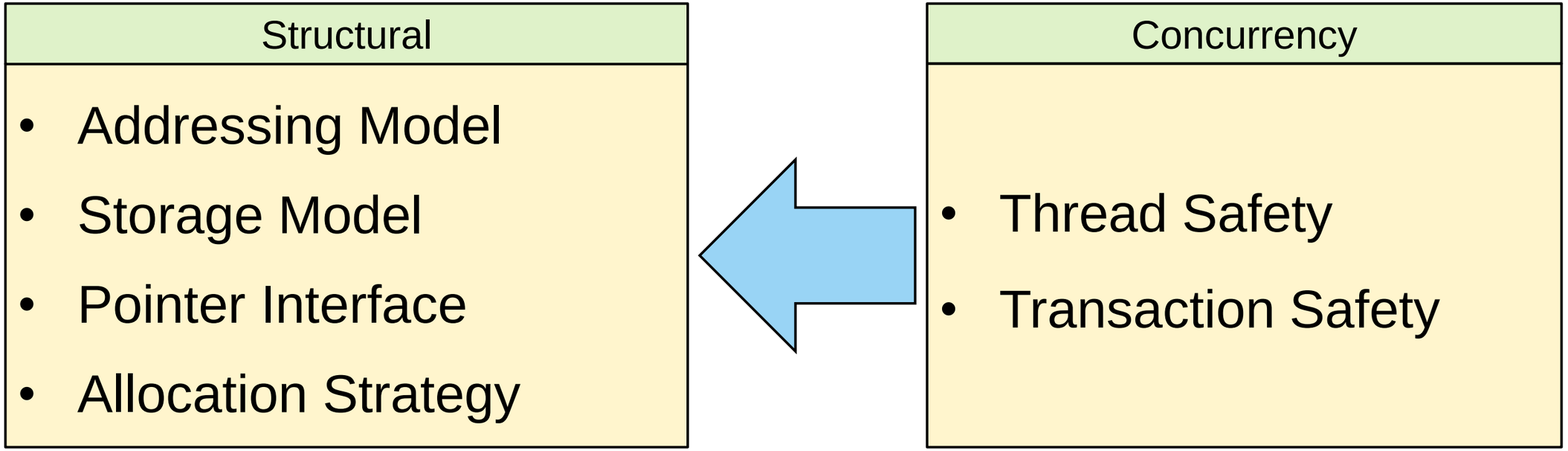
- Analogous to `malloc()` / `free()`
- Analogous to `::operator new()` / `::operator delete()`

Concepts – Thread Safety and Transaction Safety

- Thread safety – correct operation with multiple threads/processes
- Transaction safety – supporting allocate/commit/rollback semantics

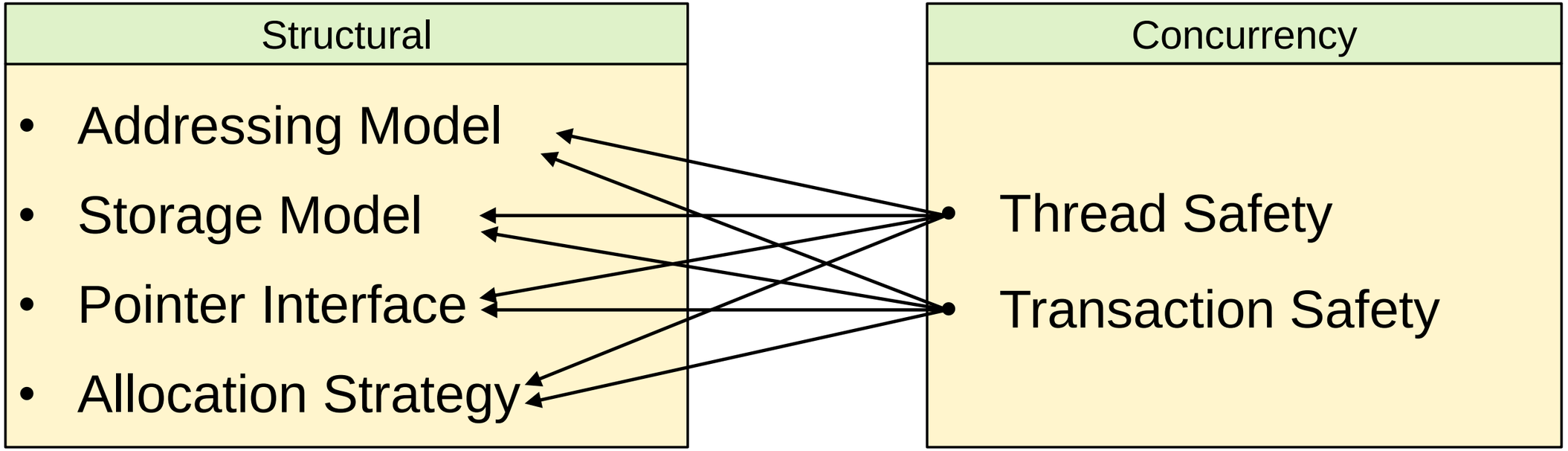
Concepts – Thread Safety and Transaction Safety

- Thread safety – correct operation with multiple threads/processes
- Transaction safety – supporting allocate/commit/rollback semantics



Concepts – Thread Safety and Transaction Safety

- Thread safety – correct operation with multiple threads/processes
- Transaction safety – supporting allocate/commit/rollback semantics



How Is `std::allocator<T>` Characterized By This Framework?

- Addressing Model: **`void*`**
- Storage Model: **`::operator new()`**
- Pointer Interface: **`T*`**
- Allocation Strategy: **`::operator new()`**
- Thread Safety: **`::operator new()`**
- Transaction Safety: *none*

Other Allocators

- dlmalloc
- jemalloc
- tcmalloc
- Hoard
- VMem

- Addressing Model: **`void*`**
- Storage Model: ***`custom`***
- Pointer Interface: **`T*`**
- Allocation Strategy: ***`custom`***
- Thread Safety: ***`custom`***
- Transaction Safety: none

Synthetic Pointer Performance Testing

Test Framework Types

- Allocator `rhx_allocator<T, AS>`

pointer \ allocator	stateless	stateful
ordinary	`std::allocator<T>`	?
synthetic	`rhx_allocator<T,AS>`	?

Test Framework Types

- Allocator `rhx_allocator<T, AS>`
- Allocation strategy `leaky_allocation_strategy<SM>`
- Pointer interface `syn_ptr<T, AM>`
- Storage model base class `storage_model_base`
- Customized addressing and storage model pairs
  - `wrapper_storage_model`
    `wrapper_addressing_model`
  - `based_2dxl_storage_model`
    `based_2dxl_addressing_model<based_2dxl_storage_model>`
  - `based_2d_storage_model`
    `based_2d_addressing_model<based_2d_storage_model>`
  - `based_1d_storage_model`
    `based_1d_addressing_model<based_1d_storage_model>`
  - `offset_storage_model`
    `offset_addressing_model<offset_storage_model>`

Test Framework Types

- **Allocator `rhx_allocator<T, AS>`**
- Allocation strategy `leaky_allocation_strategy<SM>`
- Pointer interface `syn_ptr<T, AM>`
- Storage model base class `storage_model_base`
- Customized addressing and storage model pairs
  - `wrapper_storage_model`
    `wrapper_addressing_model`
  - `based_2dxl_storage_model`
    `based_2dxl_addressing_model<based_2dxl_storage_model>`
  - `based_2d_storage_model`
    `based_2d_addressing_model<based_2d_storage_model>`
  - `based_1d_storage_model`
    `based_1d_addressing_model<based_1d_storage_model>`
  - `offset_storage_model`
    `offset_addressing_model<offset_storage_model>`

Test Allocator rhx_allocator

```
template<class T, class AS>
class rhx_allocator
{
  public:
    using propagate_on_container_copy_assignment = std::true_type;
    using propagate_on_container_move_assignment = std::true_type;
    using propagate_on_container_swap            = std::true_type;

    using difference_type       = typename AS::difference_type;
    using size_type             = typename AS::size_type;
    using void_pointer          = typename AS::void_pointer;
    using const_void_pointer    = typename AS::const_void_pointer;
    using pointer               = typename AS::template rebind_pointer<T>;
    using const_pointer         = typename AS::template rebind_pointer<T const>;
    using reference             = T&;
    using const_reference       = T const&;
    using value_type            = T;
    using element_type          = T;

    ...
};
```

Test Allocator rhx_allocator

```
template<class T, class AS>
class rhx_allocator
{
  public:
    using propagate_on_container_copy_assignment = std::true_type;
    using propagate_on_container_move_assignment = std::true_type;
    using propagate_on_container_swap            = std::true_type;

    using difference_type       = typename AS::difference_type;
    using size_type             = typename AS::size_type;
    using void_pointer          = typename AS::void_pointer;
    using const_void_pointer    = typename AS::const_void_pointer;
    using pointer               = typename AS::template rebind_pointer<T>;
    using const_pointer         = typename AS::template rebind_pointer<T const>;
    using reference             = T&;
    using const_reference       = T const&;
    using value_type            = T;
    using element_type          = T;

    ...
};
```

Test Allocator rhx_allocator

```
template<class T, class AS>
class rhx_allocator
{
    ...

    template<class U>
    struct rebind { using other = rhx_allocator<U, AS>; };

    T*          address(reference t) const noexcept;
    T const*    address(const_reference t) const noexcept;

    pointer     allocate(size_type n);
    pointer     allocate(size_type n, const_void_pointer p);
    void        deallocate(pointer p, size_type n);

    template<class U, class... Args>  void    construct(U* p, Args&&... args);
    template<class U>                 void    destroy(U* p);

  private:
    AS  m_heap;
};
```

Test Allocator `rhx_allocator` – Allocation and Deallocation

```
template<class T, class AS> inline
typename rhx_allocator<T, AS>::pointer
rhx_allocator<T, AS>::allocate(size_type n)
{
    return static_cast<pointer>(m_heap.allocate(n * sizeof(T)));
}


template<class T, class AS> inline void
rhx_allocator<T, AS>::deallocate(pointer p, size_type)
{
    m_heap.deallocate(p);
}
```

Test Allocator `rhx_allocator` – Construct and Destroy

```
template<class T, class AS>
template<class U, class... Args> inline void
rhx_allocator<T, AS>::construct(U* p, Args&&... args)
{
    ::new ((void*) p) U(std::forward<Args>(args)...);
}


template<class T, class AS>
template<class U> inline void
rhx_allocator<T, AS>::destroy(U* p)
{
    p->~U();
}
```

Test Framework Types

- Allocator `rhx_allocator<T, AS>`
- **Allocation strategy `leaky_allocation_strategy<SM>`**
- Pointer interface `syn_ptr<T, AM>`
- Storage model base class `storage_model_base`
- Customized addressing and storage model pairs
  - `wrapper_storage_model`
    `wrapper_addressing_model`
  - `based_1d_storage_model`
    `based_1d_addressing_model<based_1d_storage_model>`
  - `based_2d_storage_model`
    `based_2d_addressing_model<based_2d_storage_model>`
  - `based_2dxl_storage_model`
    `based_2dxl_addressing_model<based_2dxl_storage_model>`
  - `offset_storage_model`
    `offset_addressing_model<offset_storage_model>`

Leaky Allocation Strategy

```
template<class SM>
class leaky_allocation_strategy
{
  public:
    using storage_model      = SM;
    using addressing_model   = typename SM::addressing_model;
    using difference_type    = typename SM::difference_type;
    using size_type          = typename SM::size_type;
    using void_pointer       = syn_ptr<void, addressing_model>;
    using const_void_pointer = syn_ptr<void const, addressing_model>;

    template<class T>
    using rebind_pointer     = syn_ptr<T, addressing_model>;

  public:
    void_pointer    allocate(size_type n);
    void            deallocate(void_pointer p);

    static  void    reset_buffers();
    static  void    swap_buffers();
};
```

Leaky Allocation Strategy - Allocation

```
template<class SM>
typename leaky_allocation_strategy<SM>::void_pointer
leaky_allocation_strategy<SM>::allocate(size_type n)
{
    if (!sm_initialized)
    {
        storage_model::init_segments();
        sm_curr_segment = storage_model::first_segment_index();
        sm_curr_offset  = 64;
        sm_initialized  = true;
    }

    size_type   chunk_size   = round_up(n, 16u);
    size_type   chunk_offset = sm_curr_offset;

    if ((chunk_offset + chunk_size) > storage_model::max_segment_size())
    {
        ++sm_curr_segment;
        chunk_offset   = 64;
        sm_curr_offset = chunk_offset + chunk_size;
    }
    else
    {
        sm_curr_offset += chunk_size;
    }

     return storage_model::segment_pointer(sm_curr_segment, chunk_offset);
}
```

Leaky Allocation Strategy – Deallocation

```
template<class SM> inline void
leaky_allocation_strategy<SM>::deallocate(void_pointer)
{}
```

Test Framework Types

- Allocator `rhx_allocator<T, AS>`
- Allocation strategy `leaky_allocation_strategy<SM>`
- **Pointer interface `syn_ptr<T, AM>`**
- Storage model base class `storage_model_base`
- Customized addressing and storage model pairs
  - `wrapper_storage_model`
    `wrapper_addressing_model`
  - `based_2dxl_storage_model`
    `based_2dxl_addressing_model<based_2dxl_storage_model>`
  - `based_2d_storage_model`
    `based_2d_addressing_model<based_2d_storage_model>`
  - `based_1d_storage_model`
    `based_1d_addressing_model<based_1d_storage_model>`
  - `offset_storage_model`
    `offset_addressing_model<offset_storage_model>`

Synthetic Pointer Interface

```
template<class T, class AM>
class syn_ptr
{
  public:
    [ Special Member Functions ]

    [ Other Constructors ]

    [ Other Assignment Operators ]

    [ Conversion Operators ]

    [ Dereferencing and Pointer Arithmetic ]

    [ Helpers to Support Library Requirements ]

    [ Helpers to Support Comparison Operators ]

  private
    [ Member Data ]
};
```

Synthetic Pointer Interface – Helper Traits for SFINAE

```
template<class From, class To>
using enable_if_convertible_t =
    typename std::enable_if<std::is_convertible<From*, To*>::value, bool>::type;

template<class From, class To>
using enable_if_not_convertible_t =
    typename std::enable_if<!std::is_convertible<From*, To*>::value, bool>::type;

template<class T1, class T2>
using enable_if_comparable_t =
    typename std::enable_if<std::is_convertible<T1*, T2 const*>::value ||
                            std::is_convertible<T2*, T1 const*>::value, bool>::type;

template<class T, class U>
using enable_if_non_void_t =
    typename std::enable_if<!std::is_void<U>::value && std::is_same<T, U>::value, bool>::type;

template<class T>
using get_type_or_void_t =
    typename std::conditional<std::is_void<T>::value, void,
                              typename std::add_lvalue_reference<T>::type>::type;
```

Synthetic Pointer Interface – Nested Type Aliases

```
template<class T, class AM>
class syn_ptr
{
  public:
    //- Template rebinder for std::pointer_traits.
    //
    template<class U>  using rebind = syn_ptr<U, AM>;

    //- Other aliases required by allocator_traits<T>, pointer_traits<T>, and the containers.
    //
    using difference_type   = typename AM::difference_type;
    using size_type         = typename AM::size_type;
    using element_type      = T;
    using value_type        = T;
    using reference         = get_type_or_void_t<T>;
    using pointer           = syn_ptr;

    using iterator_category = std::random_access_iterator_tag;

    ...
};
```

Synthetic Pointer Interface – Special Member Functions

```
template<class T, class AM>
class syn_ptr
{
    ...

    //- Special member functions.
    //
    ~syn_ptr() noexcept = default;

    syn_ptr() noexcept = default;
    syn_ptr(syn_ptr&&) noexcept = default;
    syn_ptr(syn_ptr const&) noexcept = default;

    syn_ptr&  operator =(syn_ptr&&) noexcept = default;
    syn_ptr&  operator =(syn_ptr const&) noexcept = default;

    ...
};
```

Synthetic Pointer Interface – Other Constructors

```
template<class T, class AM>
class syn_ptr
{
    ...

    //- User-defined construction.  Allow only implicit conversion at compile time.
    //
    syn_ptr(AM am);
    syn_ptr(std::nullptr_t);

    template<class U, enable_if_convertible_t<U, T> = true>
    syn_ptr(U* p);

    template<class U, enable_if_convertible_t<U, T> = true>
    syn_ptr(syn_ptr<U, AM> const& p);

    ...
};
```

Synthetic Pointer Interface – Other Assignment Operators

```
template<class T, class AM>
class syn_ptr
{
    ...

    //- User-defined assignment.
    //
    syn_ptr&   operator =(std::nullptr_t);

    template<class U, enable_if_convertible_t<U, T> = true>
    syn_ptr&   operator =(U* p);

    template<class U, enable_if_convertible_t<U, T> = true>
    syn_ptr&   operator =(syn_ptr<U, AM> const& p);

    ...
};
```

Synthetic Pointer Interface – Conversion Operators

```
template<class T, class AM>
class syn_ptr
{
    ...

    //- User-defined conversion.
    //
    explicit    operator bool() const;

    template<class U, enable_if_convertible_t<T, U> = true>
                operator U* () const;

    template<class U, enable_if_not_convertible_t<T, U> = true>
    explicit    operator U* () const;

    template<class U, enable_if_not_convertible_t<T, U> = true>
    explicit    operator syn_ptr<U, AM>() const;

    ...
};
```

Synthetic Pointer Interface – De-referencing

```
template<class T, class AM>
class syn_ptr
{
    ...

    //- De-referencing and indexing.
    //
    template<class U = T, enable_if_non_void_t<T, U> = true>
    U*  operator ->() const;

    template<class U = T, enable_if_non_void_t<T, U> = true>
    U&  operator  *() const;

    template<class U = T, enable_if_non_void_t<T, U> = true>
    U&  operator [](size_type n) const;

    ...
};
```

Test Pointer Interface – Pointer Arithmetic

```
template<class T, class AM>
class syn_ptr
{
    ...

    //- Pointer arithmetic operators.
    //
    template<class U = T, enable_if_non_void_t<T, U> = true>
    difference_type operator -(const syn_ptr& p) const;

    template<class U = T, enable_if_non_void_t<T, U> = true>
    syn_ptr         operator -(difference_type n) const;

    template<class U = T, enable_if_non_void_t<T, U> = true>
    syn_ptr         operator +(difference_type n) const;

    ...
};
```

Synthetic Pointer Interface – Pointer Arithmetic

```
template<class T, class AM>
class syn_ptr
{
    ...
    template<class U = T, enable_if_non_void_t<T, U> = true>
    syn_ptr&        operator ++();
    template<class U = T, enable_if_non_void_t<T, U> = true>
    syn_ptr const   operator ++(int);

    template<class U = T, enable_if_non_void_t<T, U> = true>
    syn_ptr&        operator --();
    template<class U = T, enable_if_non_void_t<T, U> = true>
    syn_ptr const   operator --(int);

    template<class U = T, enable_if_non_void_t<T, U> = true>
    syn_ptr&        operator +=(difference_type n);
    template<class U = T, enable_if_non_void_t<T, U> = true>
    syn_ptr&        operator -=(difference_type n);
    ...
};
```

Synthetic Pointer Interface

```
template<class T, class AM>
class syn_ptr
{
    ...
    //- Helper function required by pointer_traits<T>.
    //
    template<class U = T, enable_if_non_void_t<T, U> = true>
    static  syn_ptr pointer_to(U& e);

    //- Helper functions used to implement the comparison operators.
    //
    bool    equals(std::nullptr_t) const;
    template<class U, enable_if_comparable_t<T, U> = true>

    bool    equals(U const* p) const;

    template<class U, enable_if_comparable_t<T, U> = true>
    bool    equals(syn_ptr<U, AM> const& p) const;

    //- less_than() and greater_than() go here
};
```

Synthetic Pointer Interface

```
template<class T, class AM>
class syn_ptr
{
    ...

 private:
    template<class OT, class OAM> friend class syn_ptr;   //- For parametrized conversion ctor

    AM  m_addrmodel;
};
```

Synthetic Pointer Interface - Casting

```
//  template<class U, enable_if_convertible_t<T, U> = true>
//              operator U* () const;

template<class T, class AM>
template<class U, enable_if_convertible_t<T, U>> inline
syn_ptr<T, AM>::operator U* () const
{
    return static_cast<U*>(m_addrmodel.address());
}


//  template<class U, enable_if_not_convertible_t<T, U> = true>
//  explicit    operator U* () const;

template<class T, class AM>
template<class U, enable_if_not_convertible_t<T, U>> inline
syn_ptr<T, AM>::operator U* () const
{
    return static_cast<U*>(m_addrmodel.address());
}
```

Synthetic Pointer Interface - Dereferencing

```
//  template<class U = T, enable_if_non_void_t<T, U> = true>
//  U*  operator ->() const;

template<class T, class AM>
template<class U, enable_if_non_void_t<T, U>> inline U*
syn_ptr<T, AM>::operator ->() const
{
    return static_cast<U*>(m_addrmodel.address());
}


//  template<class U = T, enable_if_non_void_t<T, U> = true>
//  U&  operator  *() const;

template<class T, class AM>
template<class U, enable_if_non_void_t<T, U>> inline U&
syn_ptr<T, AM>::operator *() const
{
    return *static_cast<U*>(m_addrmodel.address());
}
```

Test Framework Types

- Allocator `rhx_allocator<T, AS>`
- Allocation strategy `leaky_allocation_strategy<SM>`
- Pointer interface `syn_ptr<T, AM>`
- **Storage model base class `storage_model_base`**
- Customized addressing and storage model pairs
  - `wrapper_storage_model`
    `wrapper_addressing_model`
  - `based_2dxl_storage_model`
    `based_2dxl_addressing_model<based_2dxl_storage_model>`
  - `based_2d_storage_model`
    `based_2d_addressing_model<based_2d_storage_model>`
  - `based_1d_storage_model`
    `based_1d_addressing_model<based_1d_storage_model>`
  - `offset_storage_model`
    `offset_addressing_model<offset_storage_model>`

Storage Model Base Class

```
class storage_model_base
{
  public:
    using difference_type = std::ptrdiff_t;
    using size_type       = std::size_t;

    enum : size_type
    {
        max_segments = 2,           //- Don't need many for testing
        max_size     = 1u << 27     //- 128 MB segments
    };

  public:
    static  void        allocate_segment(size_type segment, size_type size = max_size);
    static  void        clear_segments();
    static  void        deallocate_segment(size_type segment);
    static  void        init_segments();
    static  void        reset_segments();
    static  void        swap_segments();
    ...
};
```

Storage Model Base Class

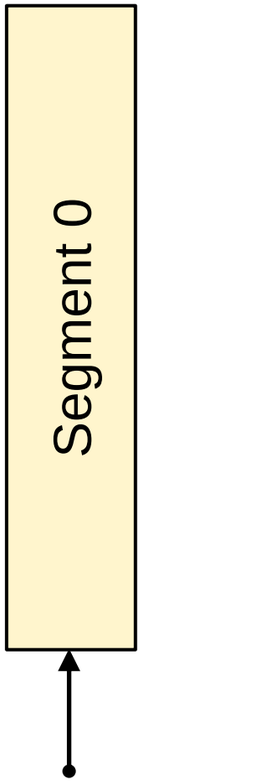
```
class storage_model_base
{
  public:
    ...

    static  char*       segment_address(size_type segment) noexcept;
    static  size_type   segment_size(size_type segment) noexcept;

    static  char*       first_segment_address() noexcept;
    static  size_type   first_segment_size() noexcept;

    static  constexpr   size_type   first_segment_index();
    static  constexpr   size_type   last_segment_index();
    static  constexpr   size_type   max_segment_count();
    static  constexpr   size_type   max_segment_size();

    ...

};
```

Storage Model Base Class

```
class storage_model_base
{
    ...

  protected:
    static  char*       sm_segment_ptrs[max_segments + 2];
    static  size_type   sm_segment_size[max_segments + 2];
    static  char*       sm_shadow_ptrs[max_segments + 2];
    static  char*&      sm_1d_base;
    static  size_type&  sm_1d_size;
    static  bool        sm_ready;
};

//------
//
inline char*
storage_model_base::segment_address(size_type segment) noexcept
{
    return sm_segment_ptrs[segment];
}
```

Storage Model Base Class – 1D Addressing View

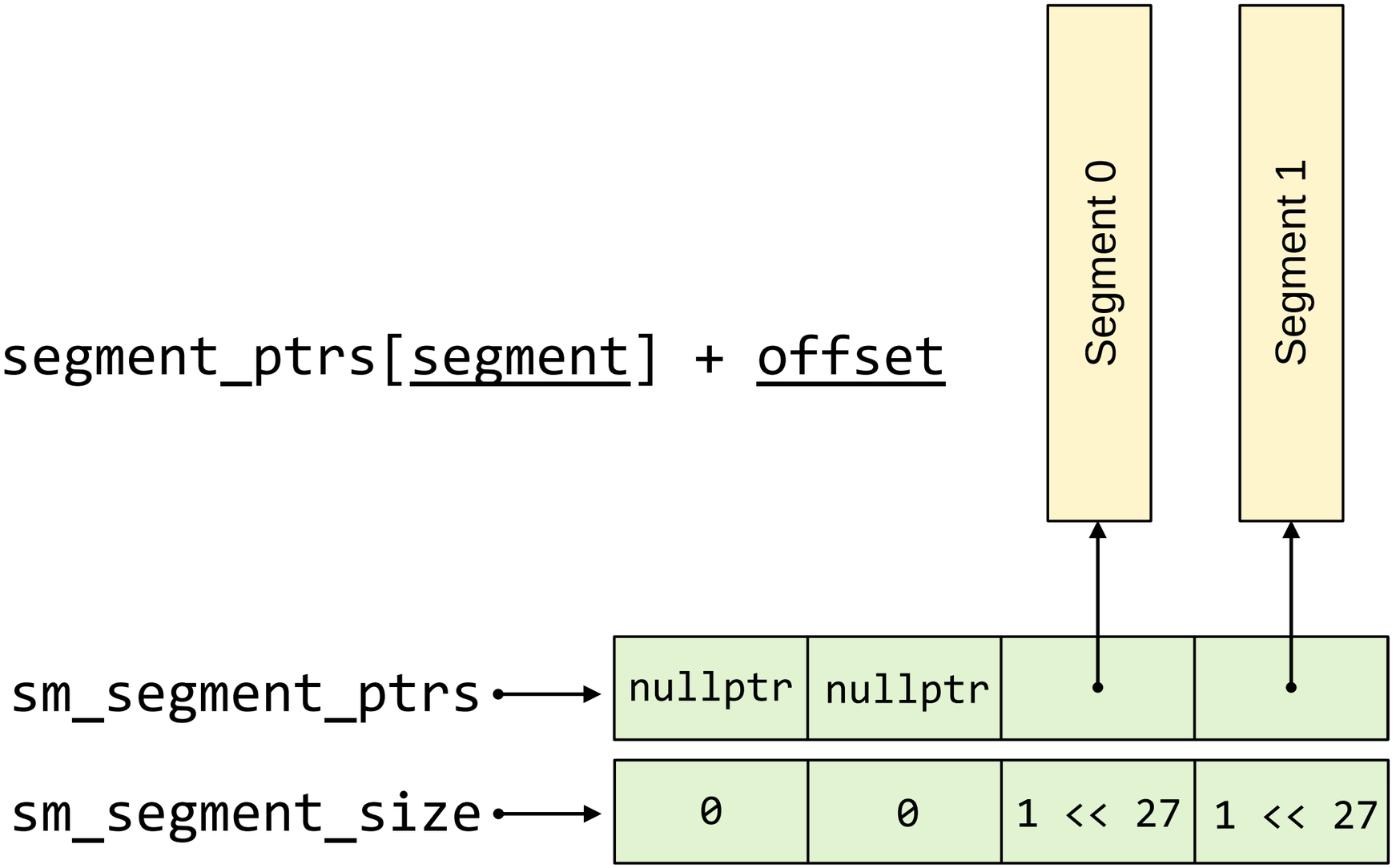
```
address = sm_1d_base + offset
```




```
sm_1d_size   1 << 27
```

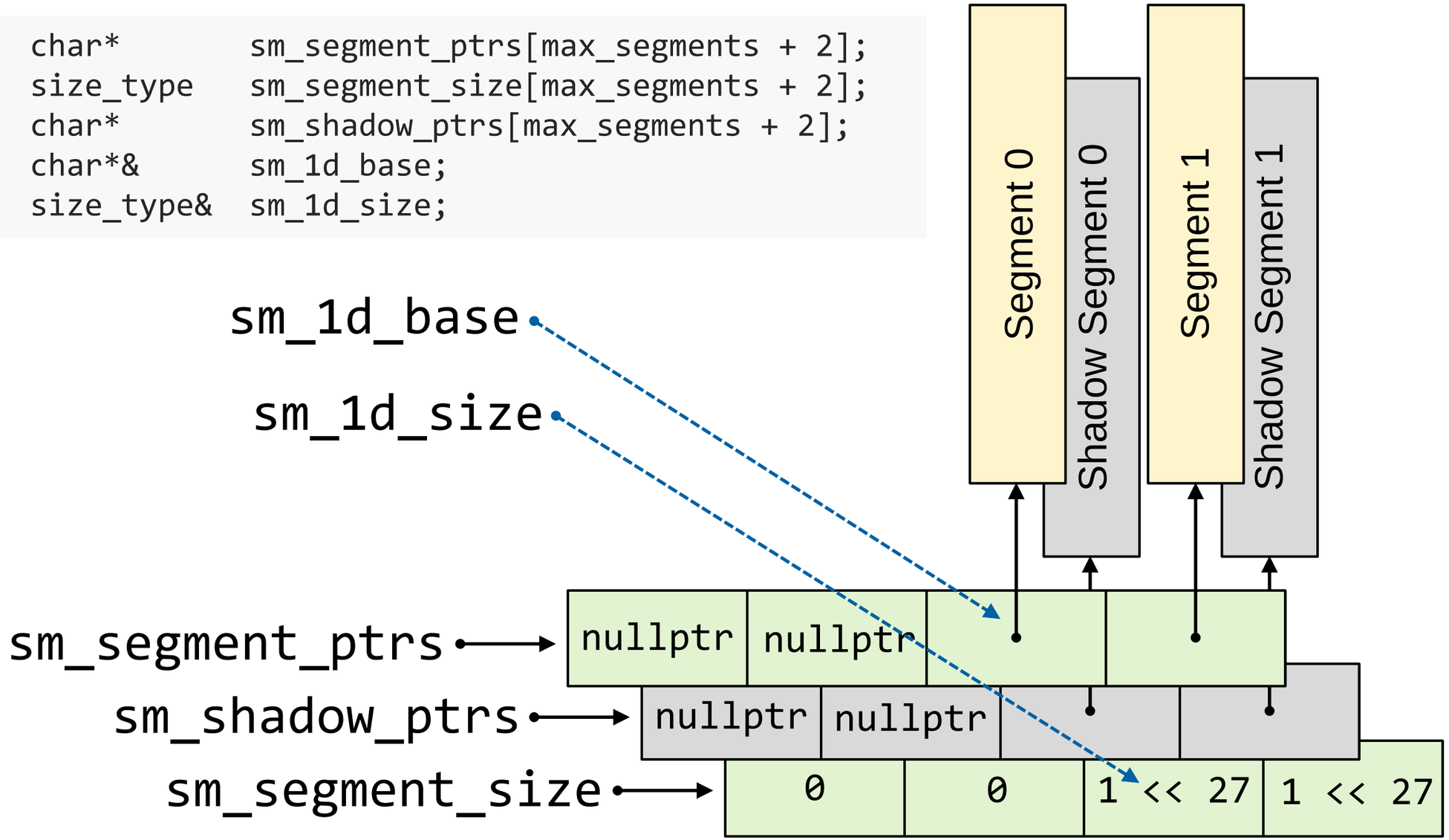
Storage Model Base Class – 2D Addressing View

```
address = sm_segment_ptrs[segment] + offset
```



Storage Model Base Class

```
static  char*       sm_segment_ptrs[max_segments + 2];
static  size_type   sm_segment_size[max_segments + 2];
static  char*       sm_shadow_ptrs[max_segments + 2];
static  char*&      sm_1d_base;
static  size_type&  sm_1d_size;
```



Test Framework Types

- Allocator `rhx_allocator<T, AS>`
- Allocation strategy `leaky_allocation_strategy<SM>`
- Pointer interface `syn_ptr<T, AM>`
- Storage model base class `storage_model_base`
- Customized addressing and storage model pairs
  - **`wrapper_storage_model`**
    `wrapper_addressing_model`
  - `based_2dxl_storage_model`
    `based_2dxl_addressing_model<based_2dxl_storage_model>`
  - `based_2d_storage_model`
    `based_2d_addressing_model<based_2d_storage_model>`
  - `based_1d_storage_model`
    `based_1d_addressing_model<based_1d_storage_model>`
  - `offset_storage_model`
    `offset_addressing_model<offset_storage_model>`

Wrapper Storage Model

```
class wrapper_storage_model : public storage_model_base
{
  public:
    using addressing_model = wrapper_addressing_model;

    static  addressing_model    segment_pointer(size_type segment, size_type offset=0);

};

//- The leaky_allocation_strategy<SM> class uses this function when allocating a chunk.
//
inline wrapper_storage_model::addressing_model
wrapper_storage_model::segment_pointer(size_type segment, size_type offset)
{
    return addressing_model{segment_address(segment) + offset};
}
```

Wrapper Addressing Model

```
class wrapper_addressing_model
{
  public:
    using size_type       = std::size_t;
    using difference_type = std::ptrdiff_t;

  public:
    ~wrapper_addressing_model() = default;

    wrapper_addressing_model() noexcept = default;
    wrapper_addressing_model(wrapper_addressing_model&&) noexcept = default;
    wrapper_addressing_model(wrapper_addressing_model const&) noexcept = default;
    wrapper_addressing_model(std::nullptr_t) noexcept;
    wrapper_addressing_model(void*) noexcept;

    wrapper_addressing_model& operator =(wrapper_addressing_model&&) noexcept = default;
    wrapper_addressing_model& operator =(wrapper_addressing_model const&) noexcept = default;
    wrapper_addressing_model& operator =(std::nullptr_t) noexcept;

    ...
};
```

Wrapper Addressing Model

```
class wrapper_addressing_model
{
    public:
    ...

    bool    equals(std::nullptr_t) const noexcept;
    bool    equals(void const* p) const noexcept;
    bool    equals(wrapper_addressing_model const& other) const noexcept;

    bool    greater_than(std::nullptr_t) const noexcept;
    bool    greater_than(void const* p) const noexcept;
    bool    greater_than(wrapper_addressing_model const& other) const noexcept;

    bool    less_than(std::nullptr_t) const noexcept;
    bool    less_than(void const* p) const noexcept;
    bool    less_than(wrapper_addressing_model const& other) const noexcept;

    ...
};
```

Wrapper Addressing Model

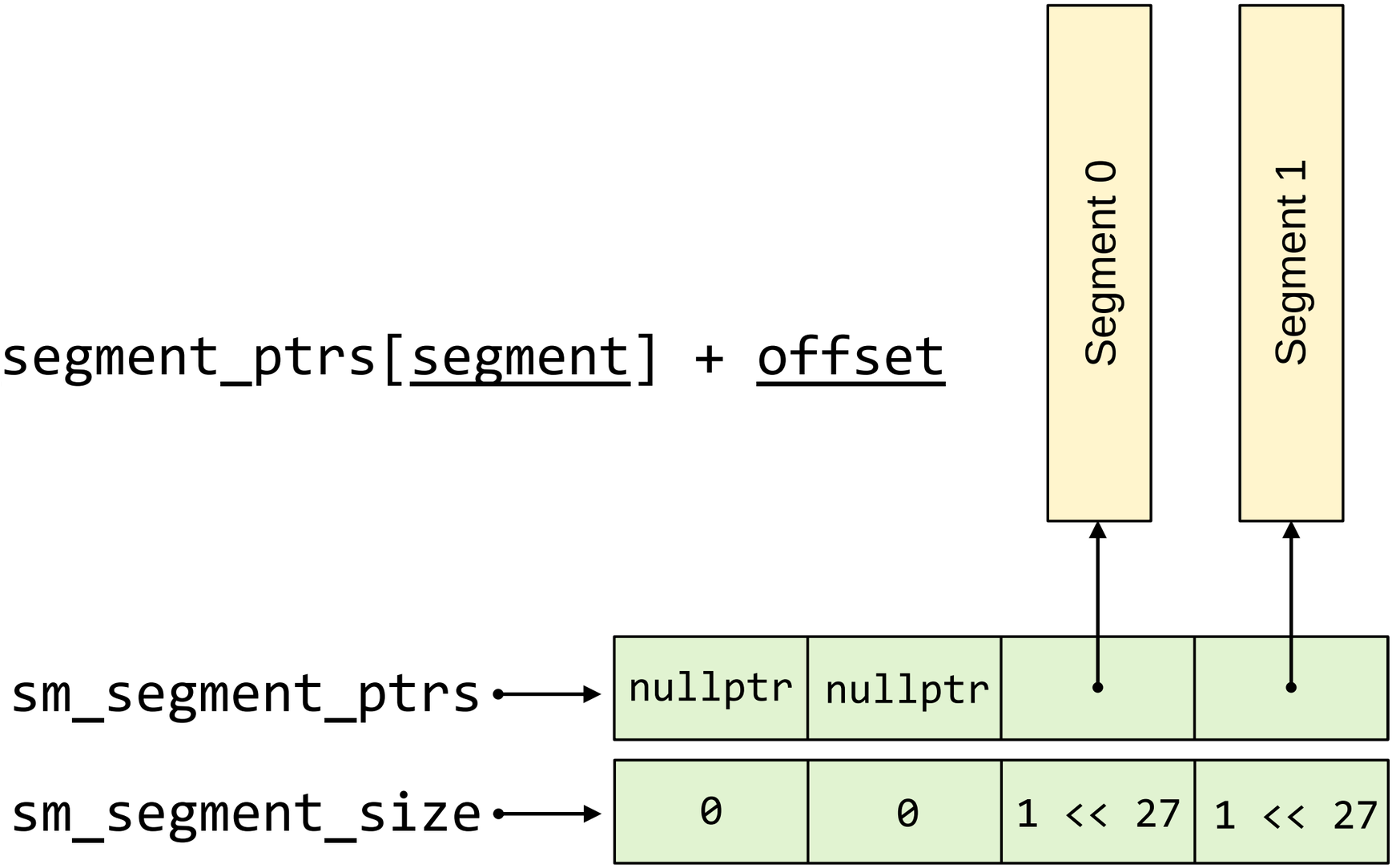
```
class wrapper_addressing_model
{
    public:
    ...

    void*   address() const noexcept;

    void    assign_from(void* p) noexcept;
    void    assign_from(void const* p) noexcept;

    void    decrement(difference_type dec) noexcept;
    void    increment(difference_type inc) noexcept;

  private:
    union
    {
        void*       m_addr;
        void const* m_caddr;
    };
};
```

Test Framework Types

- Allocator `rhx_allocator<T, AS>`
- Allocation strategy `leaky_allocation_strategy<SM>`
- Pointer interface `syn_ptr<T, AM>`
- Storage model base class `storage_model_base`
- Customized addressing and storage model pairs
  - `wrapper_storage_model`
    `wrapper_addressing_model`
  - **`based_2dxl_storage_model`**
    `based_2dxl_addressing_model<based_2dxl_storage_model>`
  - `based_2d_storage_model`
    `based_2d_addressing_model<based_2d_storage_model>`
  - `based_1d_storage_model`
    `based_1d_addressing_model<based_1d_storage_model>`
  - `offset_storage_model`
    `offset_addressing_model<offset_storage_model>`

Storage Model Base Class – 2D Addressing View

```
address = sm_segment_ptrs[segment] + offset
```



Based 2DXL Storage Model

```
class based_2dxl_storage_model : public storage_model_base
{
  public:
    using addressing_model = based_2dxl_addressing_model<based_2dxl_storage_model>;

    static  addressing_model    segment_pointer(size_type segment, size_type offset=0);
};

//------
//
inline based_2dxl_storage_model::addressing_model
based_2dxl_storage_model::segment_pointer(size_type segment, size_type offset)
{
    return addressing_model{segment, offset};
}
```

Based 2DXL Addressing Model

```
template<typename SM>
class based_2dxl_addressing_model
{
  public:
    using size_type       = std::size_t;
    using difference_type = std::ptrdiff_t;

  public:
    ~based_2dxl_addressing_model() = default;

    based_2dxl_addressing_model() noexcept = default;
    based_2dxl_addressing_model(based_2dxl_addressing_model&&) noexcept = default;
    based_2dxl_addressing_model(based_2dxl_addressing_model const&) noexcept = default;
    based_2dxl_addressing_model(std::nullptr_t) noexcept;
    based_2dxl_addressing_model(size_type segment, size_type offset) noexcept;

    based_2dxl_addressing_model&  operator =(based_2dxl_addressing_model&&) noexcept = default;
    based_2dxl_addressing_model&  operator =(based_2dxl_addressing_model const&)noexcept=default;
    based_2dxl_addressing_model&  operator =(std::nullptr_t) noexcept;
    ...
};
```

Based 2DXL Addressing Model

```
template<typename SM>
class based_2dxl_addressing_model
{
  public:
    ...

    void*   address() const noexcept;
    void    assign_from(void const* p);

    void    decrement(difference_type dec) noexcept;
    void    increment(difference_type inc) noexcept;

  private:
    uint64_t    m_offset;
    uint64_t    m_segment;
};
```

Based 2DXL Addressing Model

```
template<typename SM> inline void*
based_2dxl_addressing_model<SM>::address() const noexcept
{
    return SM::segment_address(m_segment) + m_offset;
}

template<typename SM> inline void
based_2dxl_addressing_model<SM>::increment(difference_type inc) noexcept
{
    m_offset += inc;
}
```

Based 2DXL Addressing Model

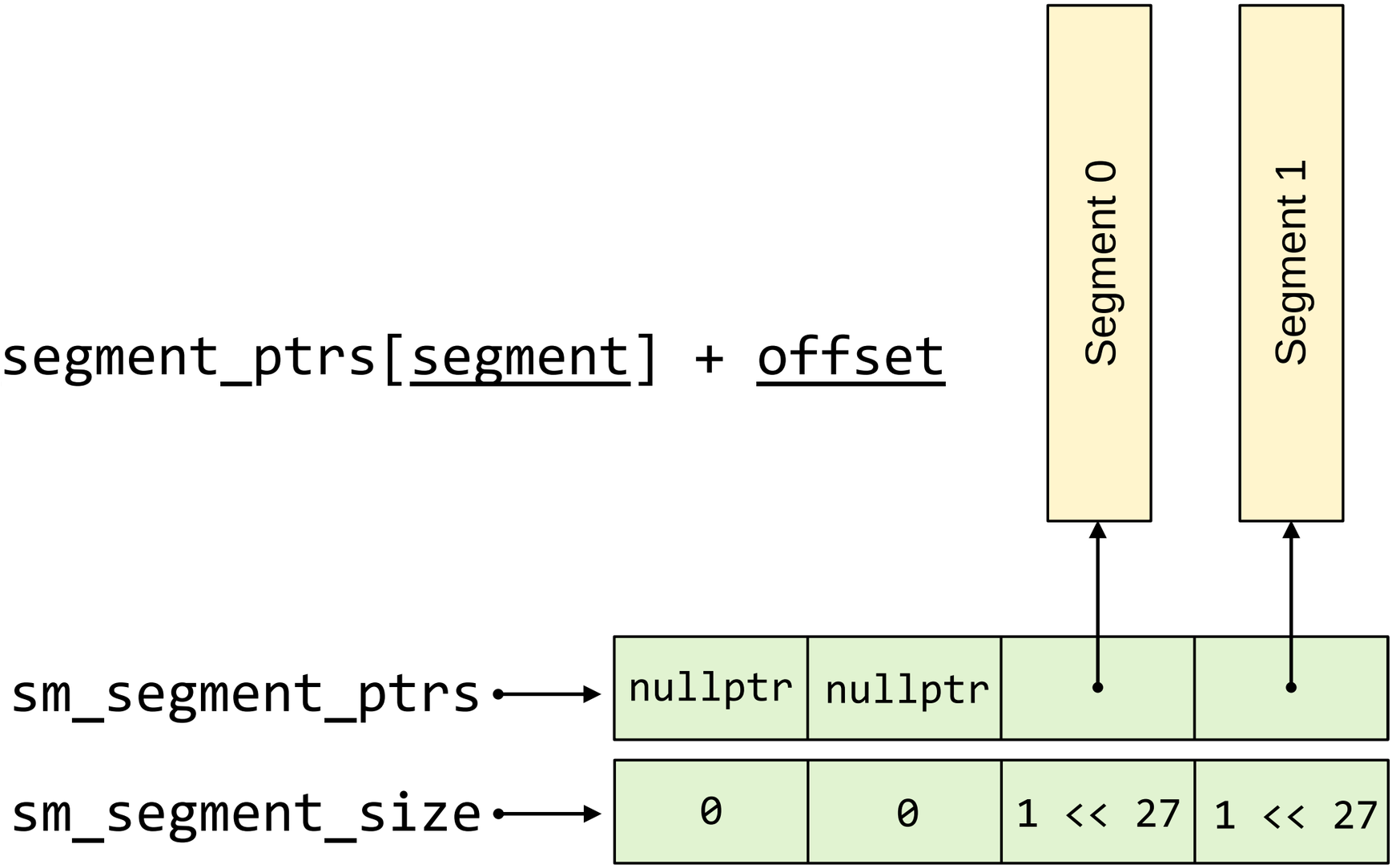
```
template<typename SM> void
based_2dxl_addressing_model<SM>::assign_from(void const* p)
{
    char const*     pdata = static_cast<char const*>(p);

    for (size_type idx = SM::first_segment_index();  idx <= SM::last_segment_index();  ++idx)
    {
        char const*     pbottom = SM::segment_address(segment_index);

        if (pbottom != nullptr)
        {
            char const*     ptop = pbottom + SM::segment_size(segment_index);

            if (pbottom <= pdata  &&  pdata < ptop)
            {
                m_offset  = pdata - pbottom;
                m_segment = idx;
                return;
            }
        }
    }
    m_segment = 0;
    m_offset  = pdata - static_cast<char const*>(nullptr);
}
```

Test Framework Types

- Allocator `rhx_allocator<T, AS>`
- Allocation strategy `leaky_allocation_strategy<SM>`
- Pointer interface `syn_ptr<T, AM>`
- Storage model base class `storage_model_base`
- Customized addressing and storage model pairs
  - `wrapper_storage_model`
    `wrapper_addressing_model`
  - `based_2dxl_storage_model`
    `based_2dxl_addressing_model<based_2dxl_storage_model>`
  - **`based_2d_storage_model`**
    `based_2d_addressing_model<based_2d_storage_model>`
  - `based_1d_storage_model`
    `based_1d_addressing_model<based_1d_storage_model>`
  - `offset_storage_model`
    `offset_addressing_model<offset_storage_model>`

Storage Model Base Class – 2D Addressing View

```
address = sm_segment_ptrs[segment] + offset
```



Based 2D Storage Model

```
class based_2d_storage_model : public storage_model_base
{
  public:
    using addressing_model = based_2d_addressing_model<based_2d_storage_model>;

    static  addressing_model    segment_pointer(size_type segment, size_type offset=0);
};

//------
//
inline based_2d_storage_model::addressing_model
based_2d_storage_model::segment_pointer(size_type segment, size_type offset)
{
    return addressing_model{segment, offset};
}
```

Based 2D Addressing Model

```
template<typename SM>
class based_2d_addressing_model
{
  public:
    using size_type       = std::size_t;
    using difference_type = std::ptrdiff_t;

  public:
    ~based_2d_addressing_model() = default;

    based_2d_addressing_model() noexcept = default;
    based_2d_addressing_model(based_2d_addressing_model&&) noexcept = default;
    based_2d_addressing_model(based_2d_addressing_model const&) noexcept = default;
    based_2d_addressing_model(std::nullptr_t) noexcept;
    based_2d_addressing_model(size_type segment, size_type offset) noexcept;

    based_2d_addressing_model&  operator =(based_2d_addressing_model&&) noexcept = default;
    based_2d_addressing_model&  operator =(based_2d_addressing_model const&) noexcept = default;
    based_2d_addressing_model&  operator =(std::nullptr_t) noexcept;
    ...
};
```

Based 2D Addressing Model

```
template<typename SM>
class based_2d_addressing_model
{
  public:
    ...

    void*   address() const noexcept;
    void    assign_from(void const* p);

    void    decrement(difference_type dec) noexcept;
    void    increment(difference_type inc) noexcept;

    ...
};
```

Based 2D Addressing Model

```
template<typename SM>
class based_2d_addressing_model
{   ...
  private:
    enum : uint64_t  { offset_mask = 0x0000'FFFF'FFFF'FFFF };

    struct addr_bits
    {
        uint16_t    m_word1;
        uint16_t    m_word2;
        uint16_t    m_word3;
        uint16_t    m_segment;
    };

  private:
    union
    {
        uint64_t    m_addr;
        addr_bits   m_bits;
    };
};
```

Based 2D Addressing Model

```
template<typename SM> inline void*
based_2d_addressing_model<SM>::address() const noexcept
{
    return SM::segment_address(m_bits.m_segment) + (m_addr & offset_mask);
}


template<typename SM> inline void
based_2d_addressing_model<SM>::increment(difference_type inc) noexcept
{
    m_addr += inc;
}
```

Based 2D Addressing Model

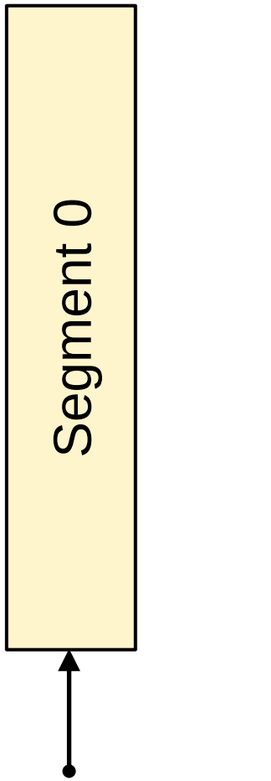
```
template<typename SM> void
based_2d_addressing_model<SM>::assign_from(void const* p)
{
    char const*     pdata = static_cast<char const*>(p);

    for (size_type i = SM::first_segment_index();  i <= SM::last_segment_index();  ++i)
    {
        char const*     pbottom = SM::segment_address(i);

        if (pbottom != nullptr)
        {
            char const*     ptop = pbottom + SM::segment_size(i);

            if (pbottom <= pdata  &&  pdata < ptop)
            {
                m_addr           = pdata - pbottom;
                m_bits.m_segment = static_cast<uint16_t>(i);
                return;
            }
        }
    }

    m_addr = pdata - static_cast<char const*>(nullptr);
}
```

Test Framework Types

- Allocator `rhx_allocator<T, AS>`
- Allocation strategy `leaky_allocation_strategy<SM>`
- Pointer interface `syn_ptr<T, AM>`
- Storage model base class `storage_model_base`
- Customized addressing and storage model pairs
  - `wrapper_storage_model`
    `wrapper_addressing_model`
  - `based_2dxl_storage_model`
    `based_2dxl_addressing_model<based_2dxl_storage_model>`
  - `based_2d_storage_model`
    `based_2d_addressing_model<based_2d_storage_model>`
  - **`based_1d_storage_model`**
    `based_1d_addressing_model<based_1d_storage_model>`
  - `offset_storage_model`
    `offset_addressing_model<offset_storage_model>`

Storage Model Base Class – 1D Addressing View

```
address = sm_1d_base + offset
```



`sm_1d_size` `1 << 27`

Based 1D Storage Model

```
class based_1d_storage_model : public storage_model_base
{
  public:
    using addressing_model = based_1d_addressing_model<based_1d_storage_model>;

    static  addressing_model    segment_pointer(size_type, size_type offset);
};

//------
//
inline based_1d_storage_model::addressing_model
based_1d_storage_model::segment_pointer(size_type, size_type offset)
{
    return addressing_model{offset};
}
```

Based 1D Addressing Model

```
template<typename SM>
class based_1d_addressing_model
{
  public:
    using size_type       = std::size_t;
    using difference_type = std::ptrdiff_t;

  public:
    ~based_1d_addressing_model() = default;

    based_1d_addressing_model() noexcept = default;
    based_1d_addressing_model(based_1d_addressing_model&&) noexcept = default;
    based_1d_addressing_model(based_1d_addressing_model const&) noexcept = default;
    based_1d_addressing_model(std::nullptr_t) noexcept;
    based_1d_addressing_model(size_type offset) noexcept;

    based_1d_addressing_model&  operator =(based_1d_addressing_model&&) noexcept = default;
    based_1d_addressing_model&  operator =(based_1d_addressing_model const&) noexcept = default;
    based_1d_addressing_model&  operator =(std::nullptr_t) noexcept;
    ...
};
```

Based 1D Addressing Model

```
template<typename SM>
class based_2d_addressing_model
{
  public:
    ...

    void*   address() const noexcept;
    void    assign_from(void const* p);

    void    decrement(difference_type dec) noexcept;
    void    increment(difference_type inc) noexcept;

    ...
};
```

Based 1D Addressing Model

```
template<typename SM>
class based_2d_addressing_model
{
    ...

  private:
    static  int const   null_offset = -1;

    int64_t     m_offset;
};
```

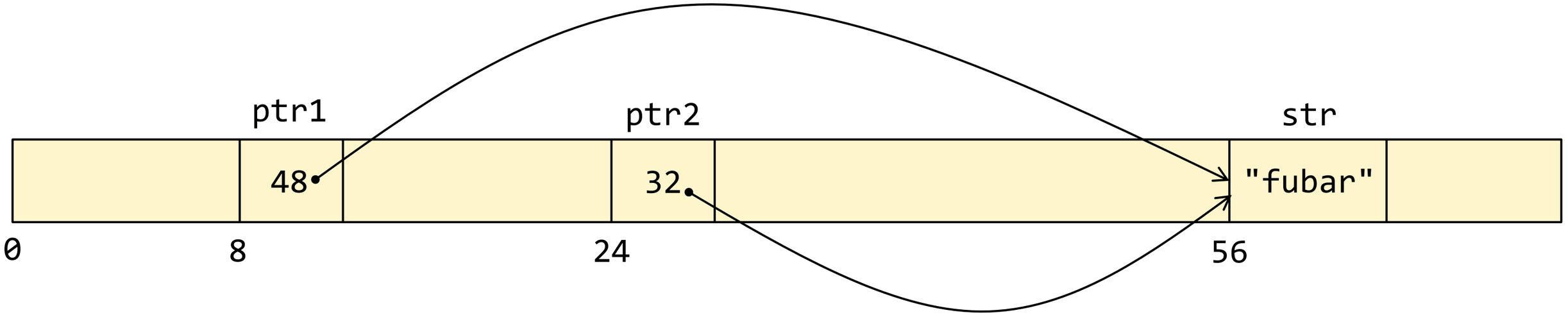
Based 1D Addressing Model

```
template<typename SM> inline void*
based_1d_addressing_model<SM>::address() const noexcept
{
    return (m_offset == null_offset) ? nullptr : SM::first_segment_address() + m_offset;
}


template<typename SM> void
based_1d_addressing_model<SM>::assign_from(void const* p)
{
    char const*     p_data  = static_cast<char const*>(p);
    char const*     p_lower = SM::first_segment_address();

    m_offset = p_data - p_lower;
}


template<typename SM> inline void
based_1d_addressing_model<SM>::increment(difference_type inc) noexcept
{
    m_offset += inc;
}
```

Test Framework Types

- Allocator `rhx_allocator<T, AS>`
- Allocation strategy `leaky_allocation_strategy<SM>`
- Pointer interface `syn_ptr<T, AM>`
- Storage model base class `storage_model_base`
- Customized addressing and storage model pairs
  - `wrapper_storage_model`
    `wrapper_addressing_model`
  - `based_2dxl_storage_model`
    `based_2dxl_addressing_model<based_2dxl_storage_model>`
  - `based_2d_storage_model`
    `based_2d_addressing_model<based_2d_storage_model>`
  - `based_1d_storage_model`
    `based_1d_addressing_model<based_1d_storage_model>`
  - **`offset_storage_model`**
    `offset_addressing_model<offset_storage_model>`

Storage Model Base Class – Offset Addressing View




```
address = (char*)this + offset
```

```
addressof(*ptr1) == addressof(*ptr2)
```

```
memcmp(&ptr1, &ptr2, sizeof(ptr1)) != 0
```

Offset Storage Model

```
class offset_storage_model : public storage_model_base
{
  public:
    using addressing_model = offset_addressing_model;

    static  addressing_model    segment_pointer(size_type, size_type offset);
};

//------
//
inline offset_storage_model::addressing_model
offset_storage_model::segment_pointer(size_type segment, size_type offset)
{
    return addressing_model{segment_address(segment) + offset};
}
```

Offset Addressing Model

```
class offset_addressing_model
{
  public:
    using size_type       = std::size_t;
    using difference_type = std::ptrdiff_t;

  public:
    ~offset_addressing_model() = default;

    offset_addressing_model() noexcept;
    offset_addressing_model(offset_addressing_model&& other) noexcept;
    offset_addressing_model(offset_addressing_model const& other) noexcept;
    offset_addressing_model(std::nullptr_t) noexcept;
    offset_addressing_model(void const* p) noexcept;

    offset_addressing_model&  operator =(offset_addressing_model&& rhs) noexcept;
    offset_addressing_model&  operator =(offset_addressing_model const& rhs) noexcept;
    offset_addressing_model&  operator =(std::nullptr_t) noexcept;
    offset_addressing_model&  operator =(void const* p) noexcept;
    ...
};
```

Offset Addressing Model

```
class offset_addressing_model
{
  public:
    ...

    void*   address() const noexcept;
    void    assign_from(void const* p);

    void    decrement(difference_type dec) noexcept;
    void    increment(difference_type inc) noexcept;

    ...
};
```

Offset Addressing Model

```
class offset_addressing_model
{
    ...

  private:
    using diff_type = difference_type ;

    enum : diff_type { null_offset = 1 };

    diff_type   m_offset;

    static  diff_type   offset_between(void const *from, void const *to) noexcept;

    diff_type   offset_to(offset_addressing_model const &other) noexcept;
    diff_type   offset_to(void const *other) noexcept;
};
```

Offset Addressing Model

```
inline offset_addressing_model::difference_type
offset_addressing_model::offset_between(void const *from, void const *to) noexcept
{
    return reinterpret_cast<intptr_t>(to) - reinterpret_cast<intptr_t>(from);
}


inline offset_addressing_model::difference_type
offset_addressing_model::offset_to(offset_addressing_model const &other) noexcept
{
    return (other.m_offset == null_offset) ? null_offset
                                           : (offset_between(this, &other) + other.m_offset);
}


inline offset_addressing_model::difference_type
offset_addressing_model::offset_to(void const *other) noexcept
{
    return (other == nullptr) ? null_offset : offset_between(this, other);
}
```

Offset Addressing Model

```
inline void*
offset_addressing_model::address() const noexcept
{
    return (m_offset == null_offset) ? nullptr :
                          reinterpret_cast<void*>(reinterpret_cast<uintptr_t>(this) + m_offset);
}


inline void
offset_addressing_model::assign_from(void const* p)
{
    m_offset = offset_to(p);
}


inline void
offset_addressing_model::increment(difference_type inc) noexcept
{
    m_offset += inc;
}
```

Offset Addressing Model

```
inline
offset_addressing_model::offset_addressing_model() noexcept
:   m_offset{null_offset}
{}


inline
offset_addressing_model::offset_addressing_model(offset_addressing_model&& rhs) noexcept
:   m_offset{offset_to(rhs)}
{}


inline
offset_addressing_model::offset_addressing_model(offset_addressing_model const& rhs) noexcept
:   m_offset{offset_to(rhs)}
{}
```

Offset Addressing Model

```
inline offset_addressing_model&
offset_addressing_model::operator =(offset_addressing_model&& rhs) noexcept
{
    m_offset = offset_to(rhs);
    return *this;
}


inline offset_addressing_model&
offset_addressing_model::operator =(offset_addressing_model const& rhs) noexcept
{
    m_offset = offset_to(rhs);
    return *this;
}
```

Synthetic Pointer Performance Testing

```
//- Let's define some strategy types.
//
using wrapper_strategy    = leaky_allocation_strategy<wrapper_storage_model>;
using based_2d_strategy   = leaky_allocation_strategy<based_2d_storage_model>;
using based_2dxl_strategy = leaky_allocation_strategy<based_2dxl_storage_model>;
using based_1d_strategy   = leaky_allocation_strategy<based_1d_storage_model>;
using offset_strategy     = leaky_allocation_strategy<offset_storage_model>;


//  class leaky_allocation_strategy<offset_storage_mode>
//  {
//    public:
//      using storage_model      = offset_storage_model;
//      using addressing_model   = offset_storage_model::addressing_model;
//      using void_pointer       = syn_ptr<void, addressing_model>;
//      using const_void_pointer = syn_ptr<void const, addressing_model>;
//
//      void_pointer    allocate(size_type n);
//      ...
//  };
```

Synthetic Pointer Performance Testing

```
template<typename AllocStrategy, typename DataType> void
run_pointer_copy_tests(char const* stype, char const* dtype);

template<typename AllocStrategy, typename DataType> void
run_pointer_sort_tests(char const* stype, char const* dtype);

template<typename AllocStrategy, typename DataType> void
run_pointer_stable_sort_tests(char const* stype, char const* dtype);


#define RUN_COPY_TESTS(ST, DT)          run_pointer_copy_tests<ST,DT>(#ST, #DT)
#define RUN_SORT_TESTS(ST, DT)          run_pointer_sort_tests<ST,DT>(#ST, #DT)
#define RUN_STABLE_SORT_TESTS(ST, DT)   run_pointer_stable_sort_tests<ST,DT>(#ST, #DT)


struct test_struct
{
    uint64_t    m1;
    uint64_t    m2;
    char        m3[48];

    test_struct();
    test_struct(test_struct const& other);
};
```

Synthetic Pointer Performance Testing

```
void run_pointer_tests()
{
    RUN_COPY_TESTS(wrapper_strategy, uint32_t);                //- Custom version of copy()
    RUN_COPY_TESTS(wrapper_strategy, uint64_t);
    RUN_COPY_TESTS(wrapper_strategy, string);
    RUN_COPY_TESTS(wrapper_strategy, test_struct);

    //- Repeat for based_2dxl_strategy, based_2d_strategy, based_1d_strategy, offset_strategy.

    RUN_SORT_TESTS(wrapper_strategy, uint32_t);                //- std::sort()
    RUN_SORT_TESTS(wrapper_strategy, uint64_t);
    RUN_SORT_TESTS(wrapper_strategy, string);
    RUN_SORT_TESTS(wrapper_strategy, test_struct);

    //- Repeat for based_2dxl_strategy, based_2d_strategy, based_1d_strategy, offset_strategy.

    RUN_STABLE_SORT_TESTS(wrapper_strategy, uint32_t);         //- std::stable_sort()
    RUN_STABLE_SORT_TESTS(wrapper_strategy, uint64_t);
    RUN_STABLE_SORT_TESTS(wrapper_strategy, string);
    RUN_STABLE_SORT_TESTS(wrapper_strategy, test_struct);

    //- Repeat for based_2dxl_strategy, based_2d_strategy, based_1d_strategy, offset_strategy.
}
```

Synthetic Pointer Performance Testing – Copying

```
template<typename II, typename OI> void
test_copy_imp(II src_begin, II src_end, OI dst_begin, OI dst_end, random_access_iterator_tag)
{
    //- Make sure the compiler doesn't try to optimize the loop into something like memcpy().
    //
    static  volatile uint64_t   dummy = 0;

    //- Make sure the sizes match;
    //
    CHECK((src_end - src_begin) == (dst_end - dst_begin));

    //- Assume the memory is contiguous and get actual pointers as the begin/end source iters.
    //
    auto const*     tmp_begin = std::addressof(*src_begin);
    auto const*     tmp_end   = tmp_begin + (src_end - src_begin);

    for (;  tmp_begin != tmp_end;  ++tmp_begin, ++dst_begin)
    {
        *dst_begin = *tmp_begin;
        ++dummy;
    }
}
```

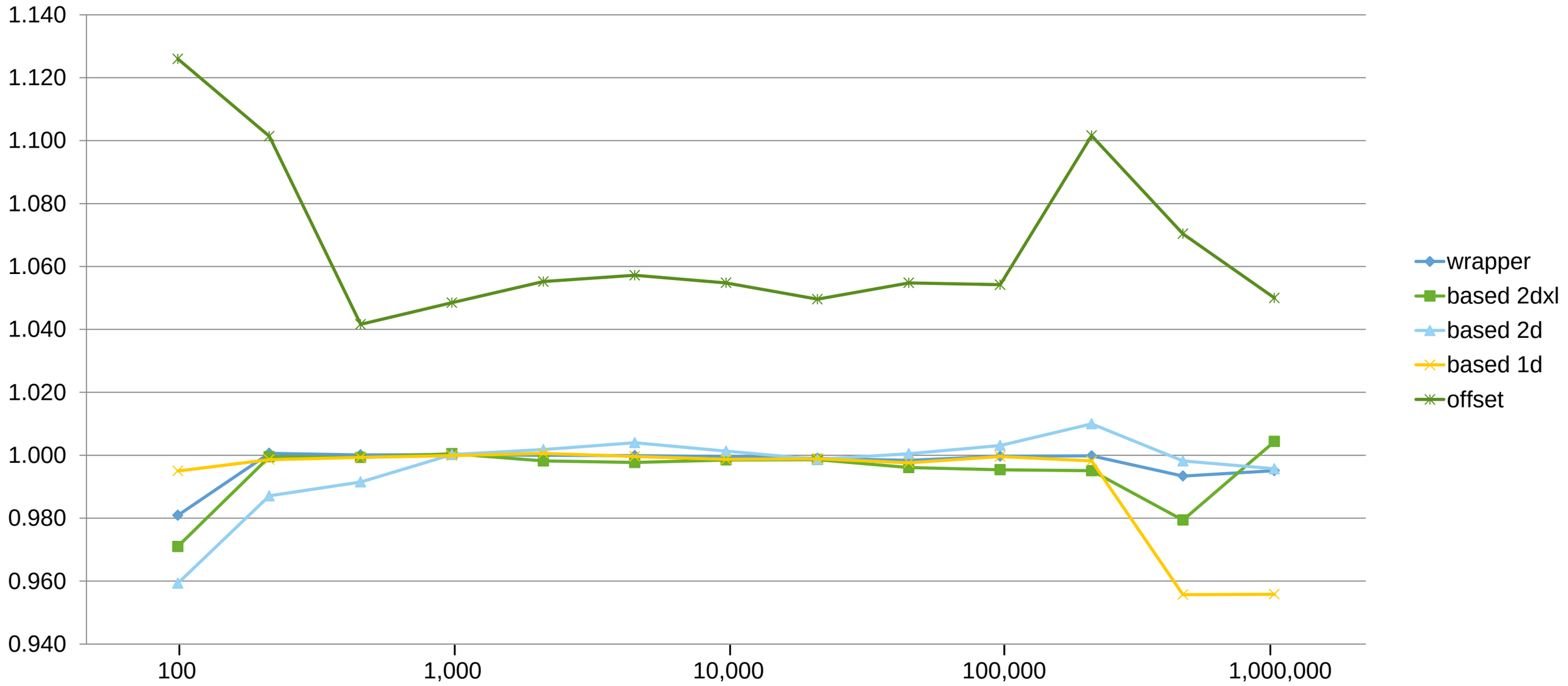
Synthetic Pointer Performance Test Outline

- 5 addressing models
  - `wrapper`, `based_2dxl`, `based_2d`, `based_1d`, `offset`
- 4 data types
  - `uint32_t`, `uint64_t`, `string`, `test_struct`
- 13 array sizes
  - 100, 200, 500, 1000, 2000, 5000, …, 1000000
- 3 algorithms
  - `copy()`, `sort()`, `stable_sort()`
- 8 compilers
  - GCC 5.4 / 6.3/ 7.1 with libstdc++
  - Clang 3.81 / 3.91 / 4.00 with libc++
  - VC++ 2015 u3 / 2017

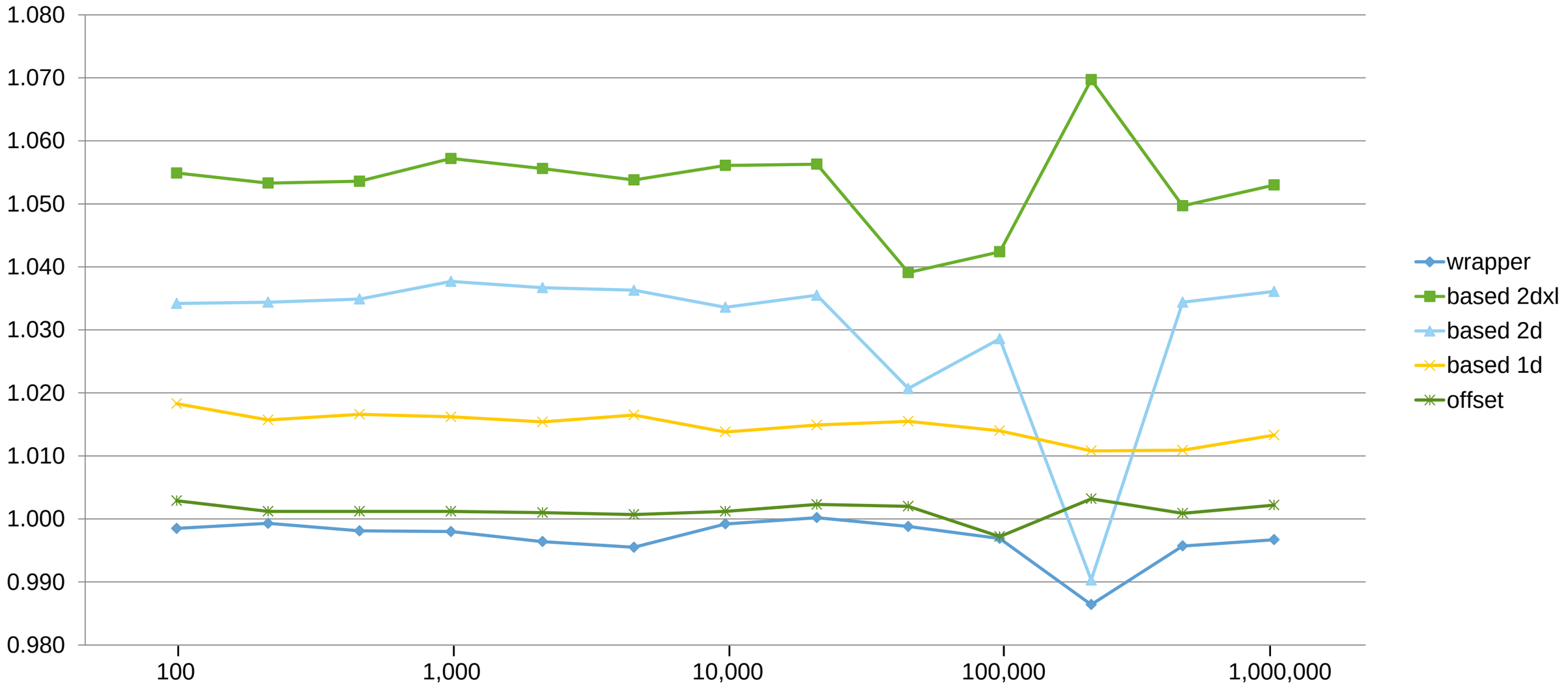
Synthetic Pointer Performance Test Procedure

- Xeon w3690, 24GB RAM
- Windows 8.1 for VS
- Centos 7.2 on VMware 11.1 for GCC/Clang
- Operations were timed using `std::chrono`
- Copy operations were repeated for a total of 10,000,000 copies
  - E.g., copying 1,000,000 elements was done 10 times
- Sorts performed 16 times, highest/lowest 3 dropped for total of 10
- All tests performed in a single thread
- All results are **ratios** – time_for_syn_ptr / time_for_ordinary_ptr

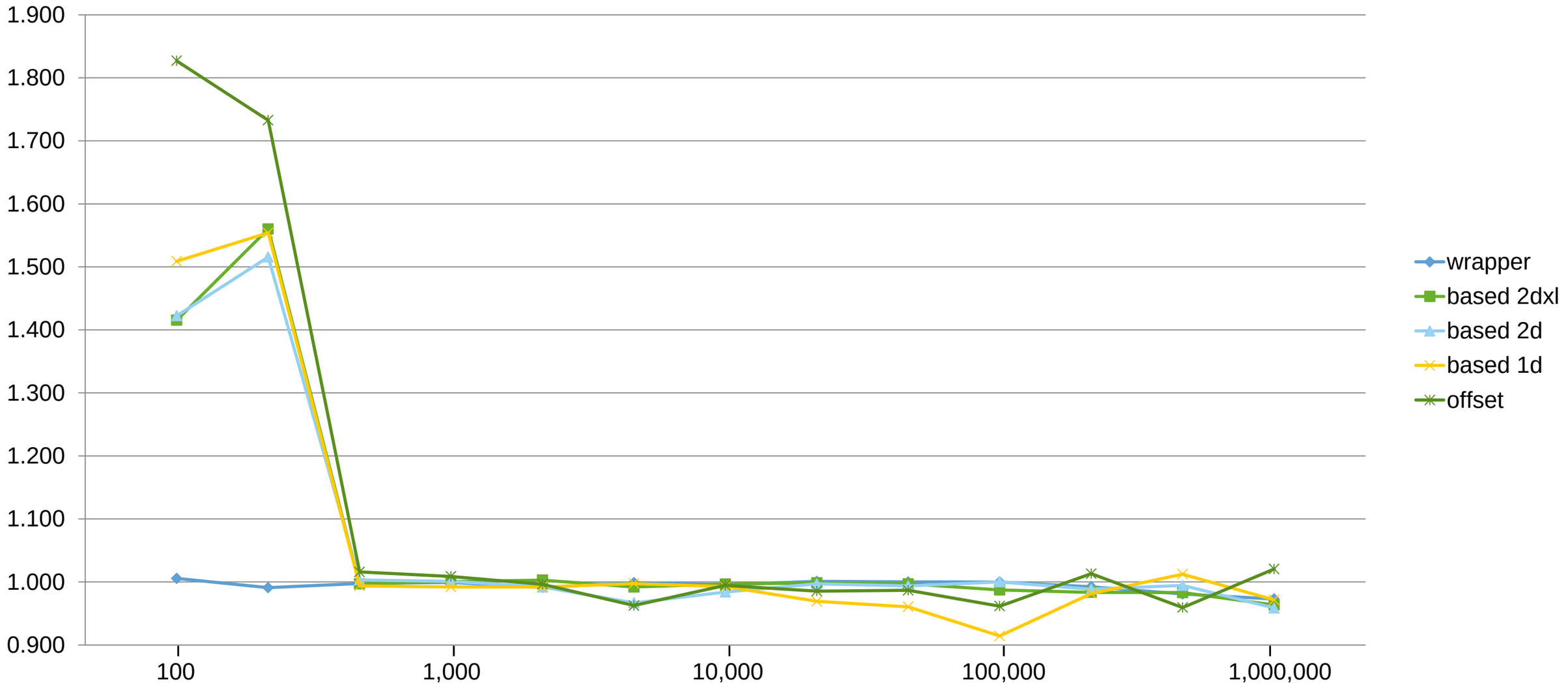
Clang 4.00 / uint64_t / copy()



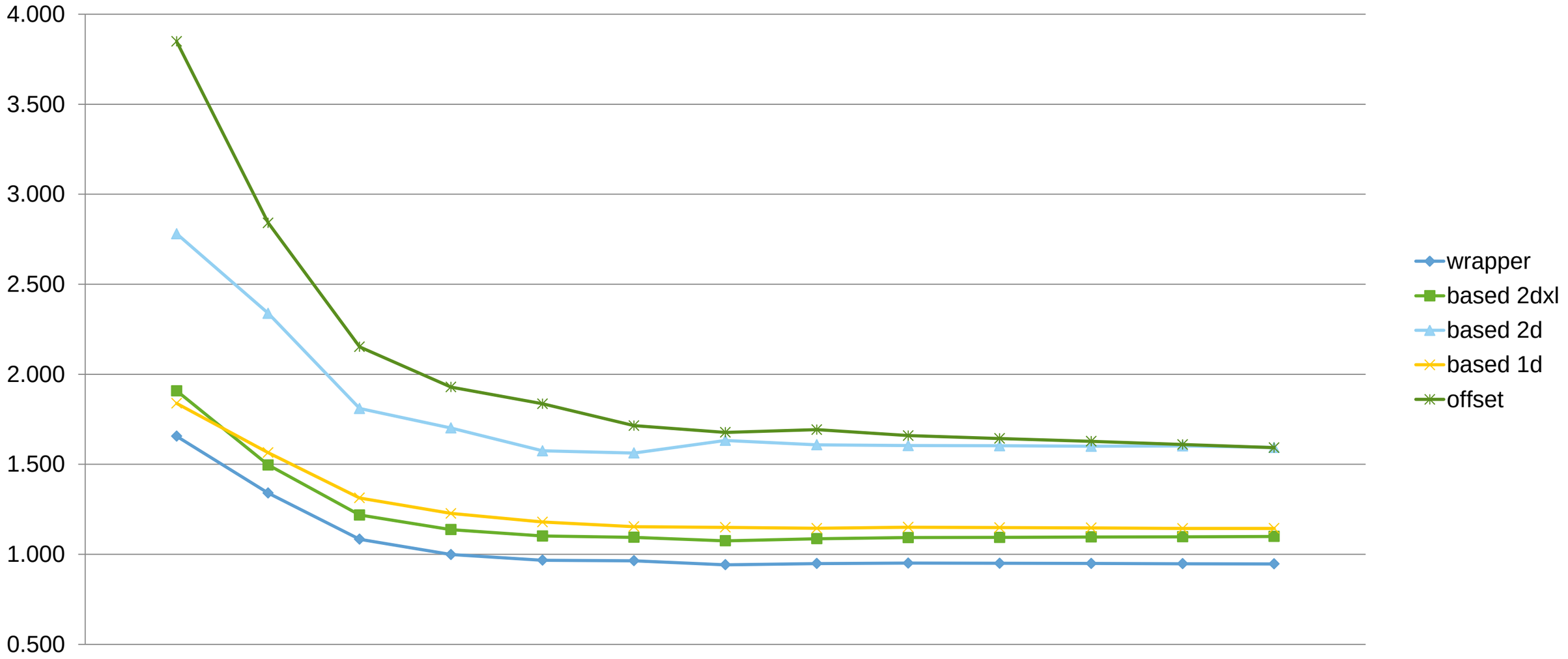
Clang 4.00 / string / copy()



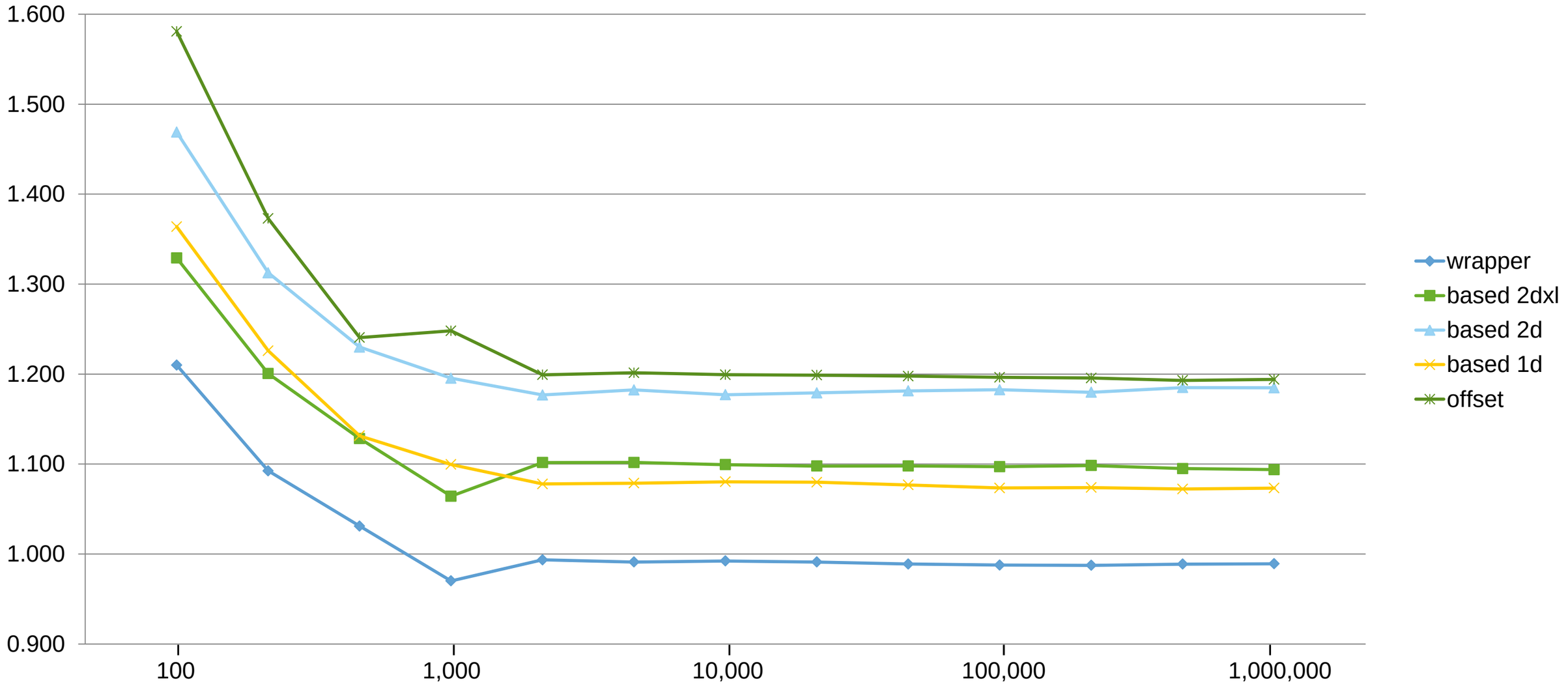
Clang 4.00 / test_struct / copy()



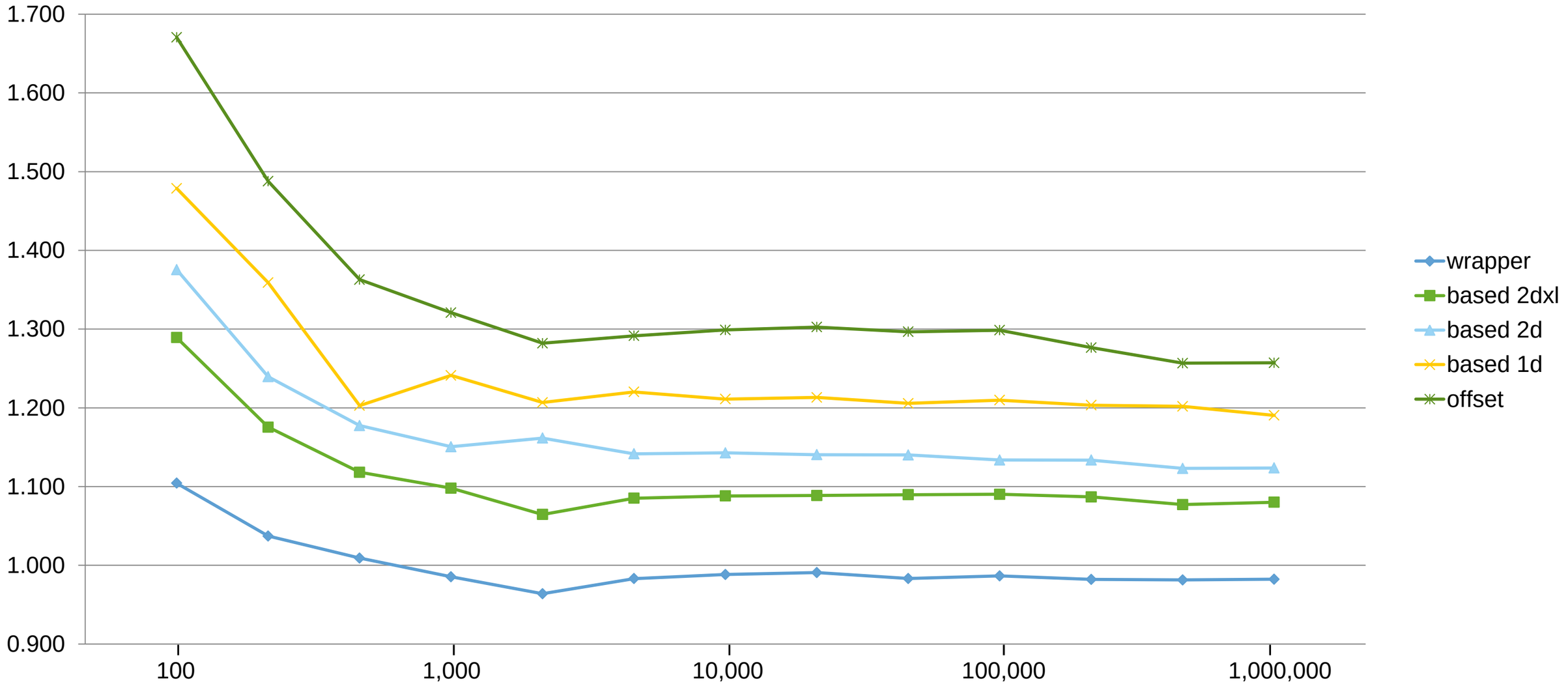
Clang 4.00 / uint64_t / sort()



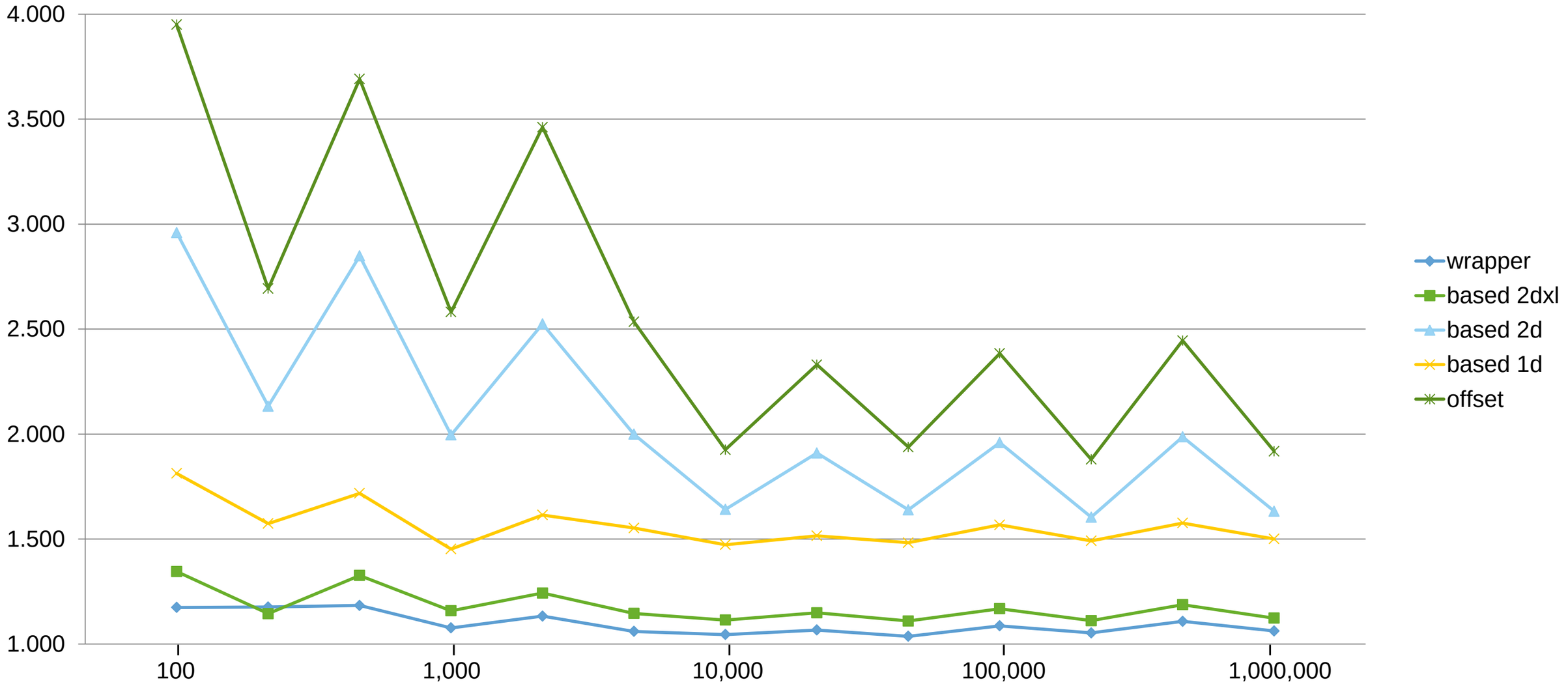
Clang 4.00 / string / sort()



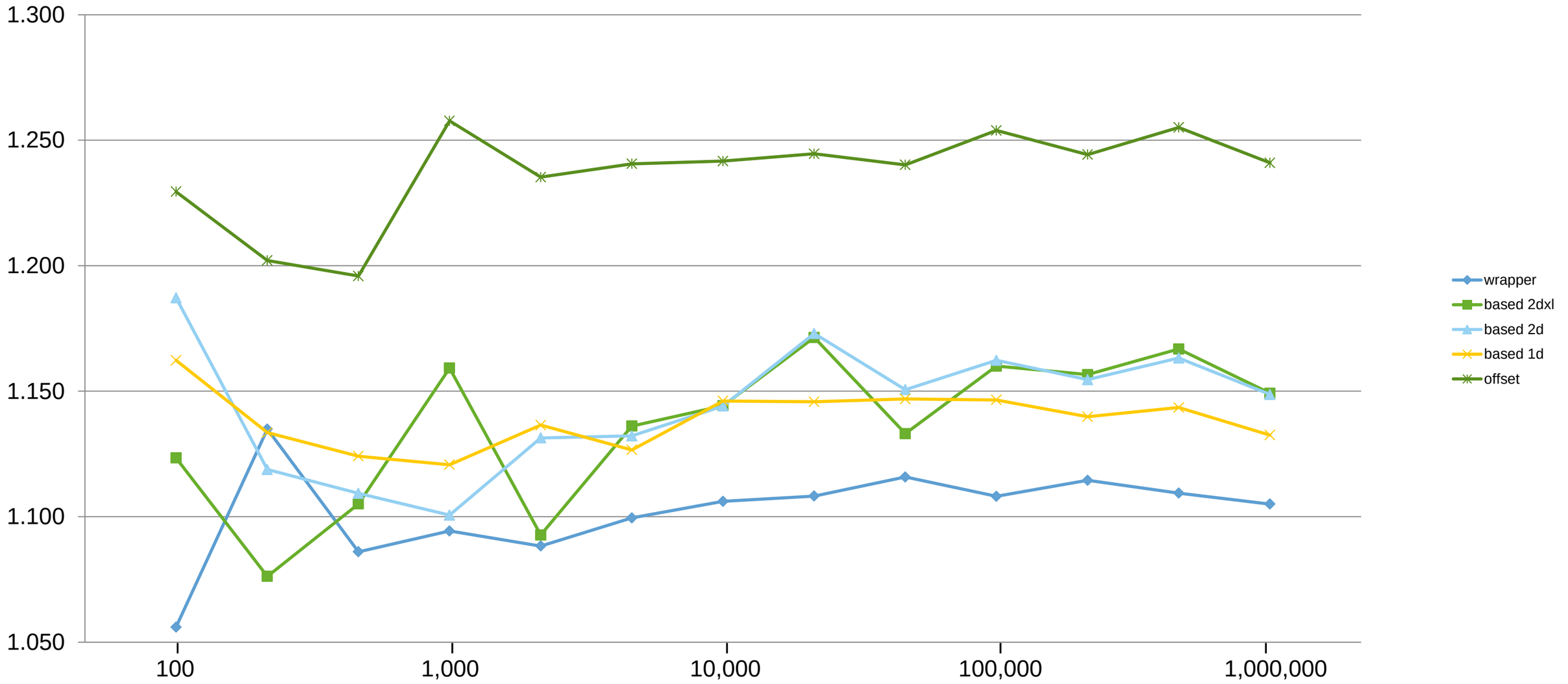
Clang 4.00 / test_struct / sort()



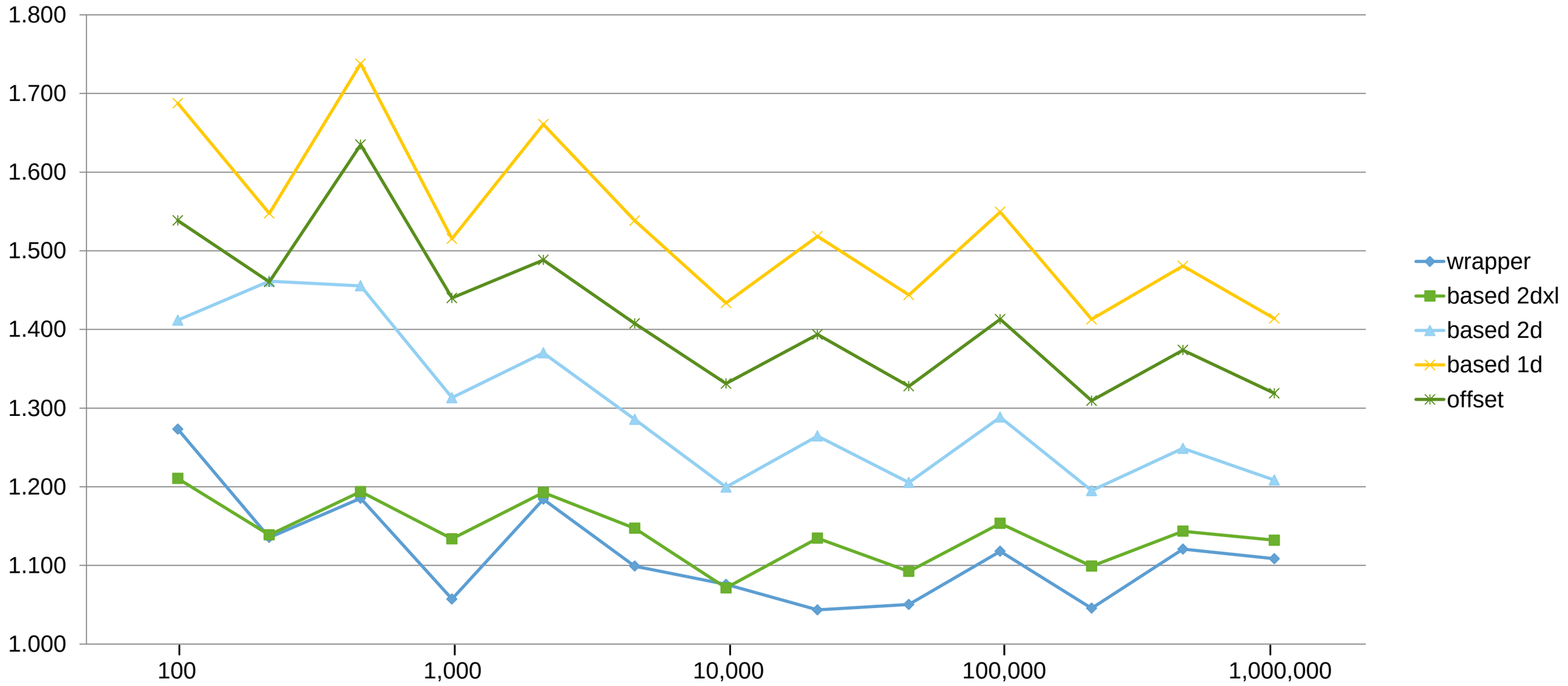
Clang 4.00 / uint64_t / stable_sort()



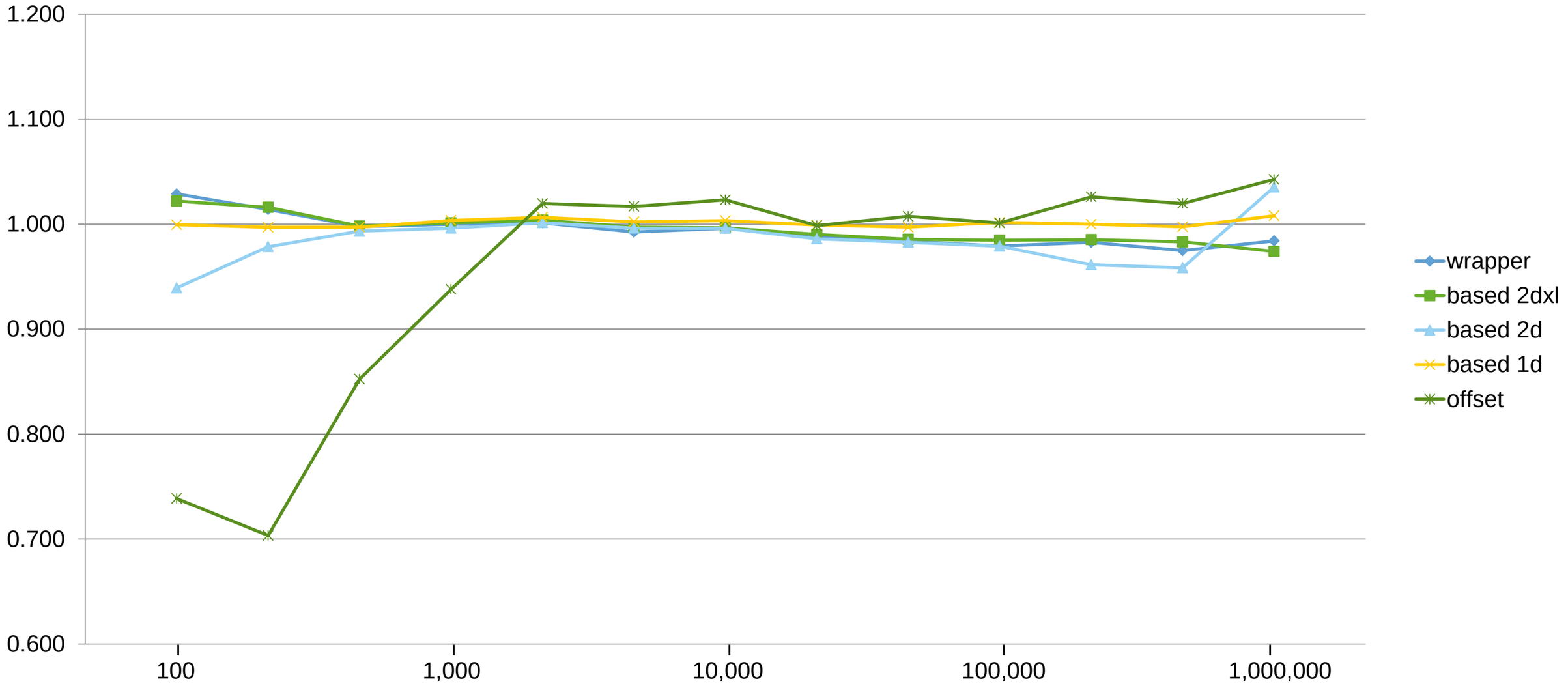
Clang 4.00 / string / stable_sort()



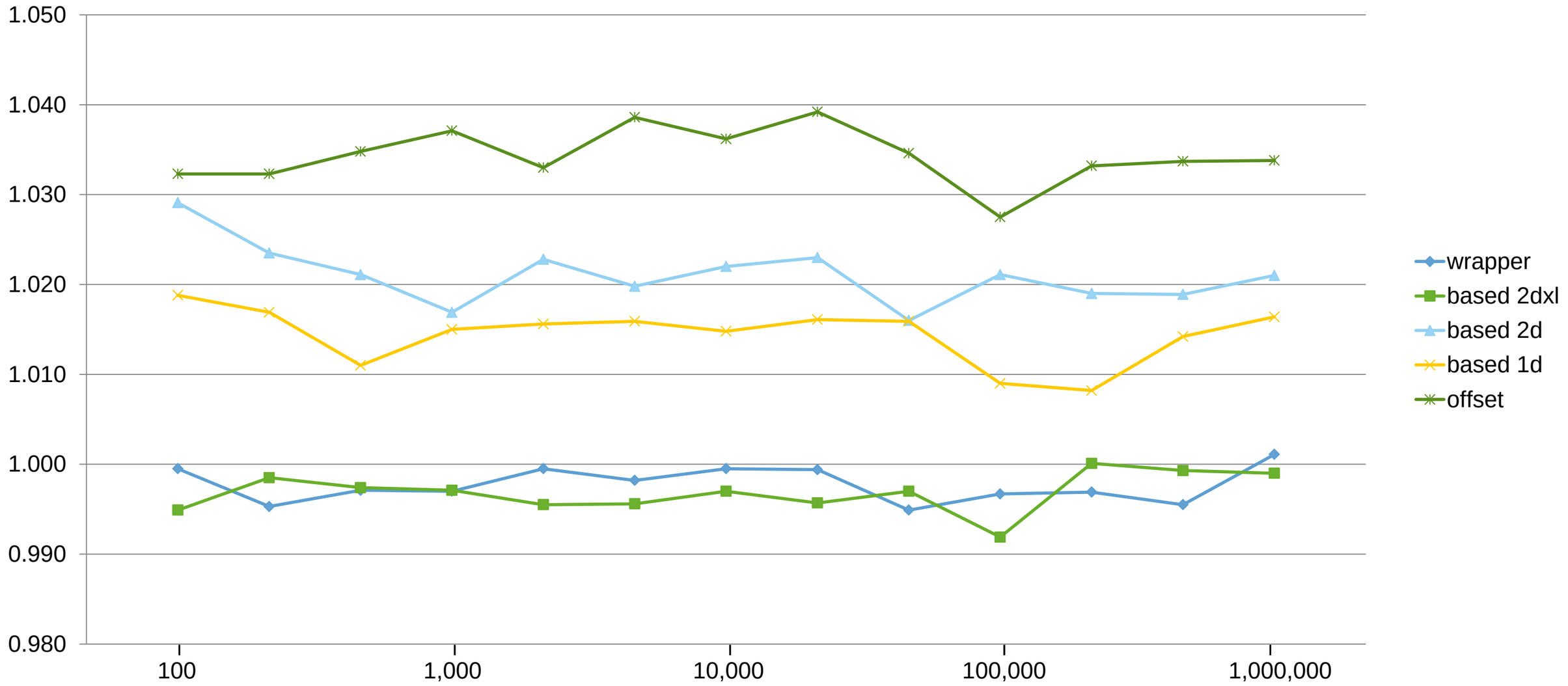
Clang 4.00 / test_struct / stable_sort()



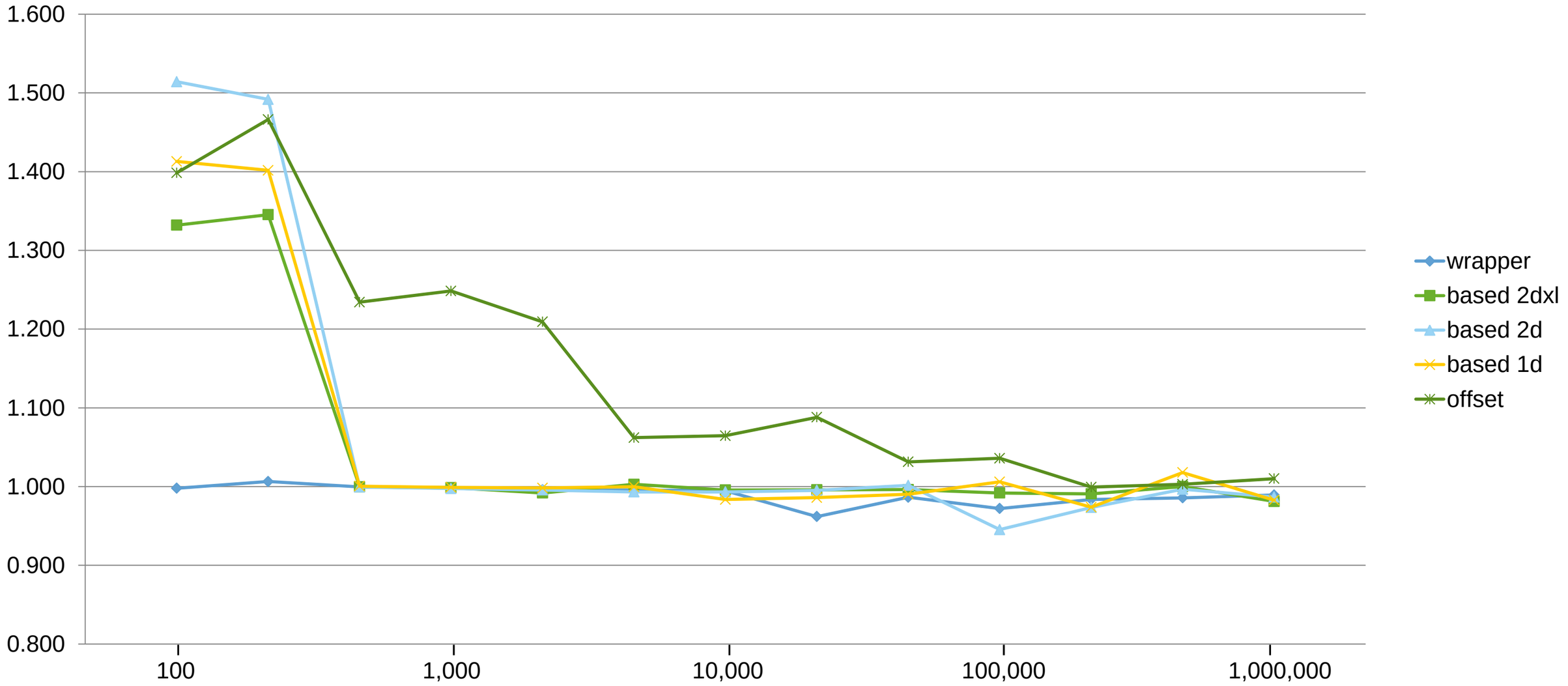
GCC 7.1 / uint64_t / copy()



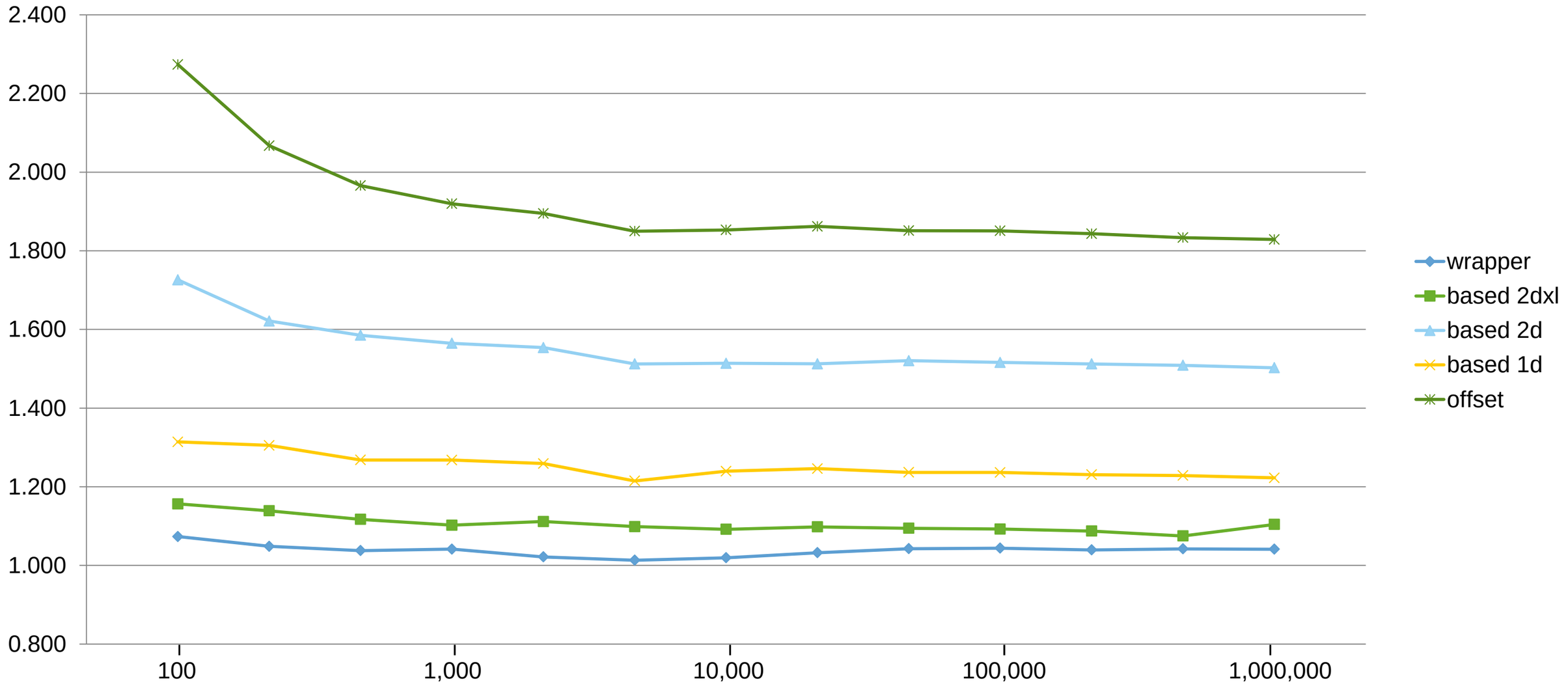
GCC 7.1 / string / copy()



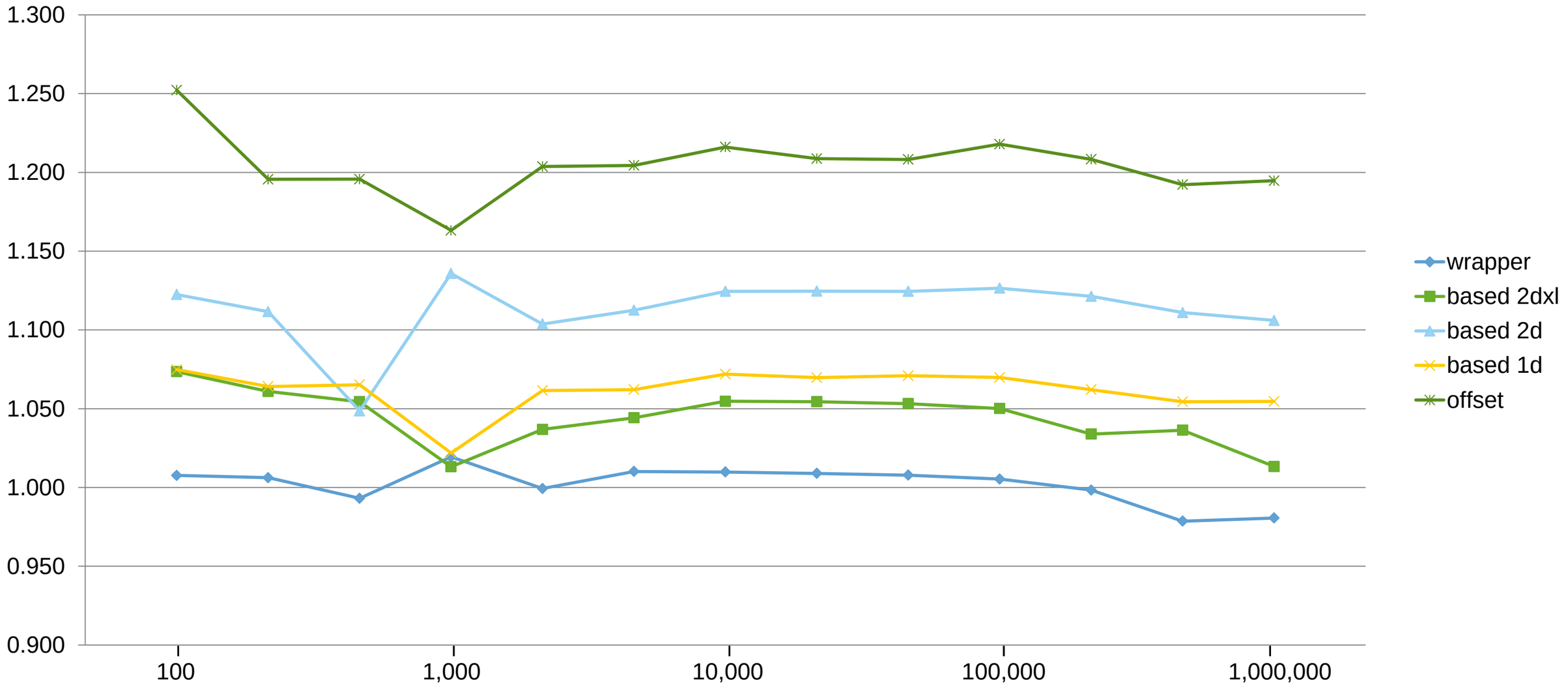
GCC 7.1 / test_struct / copy()



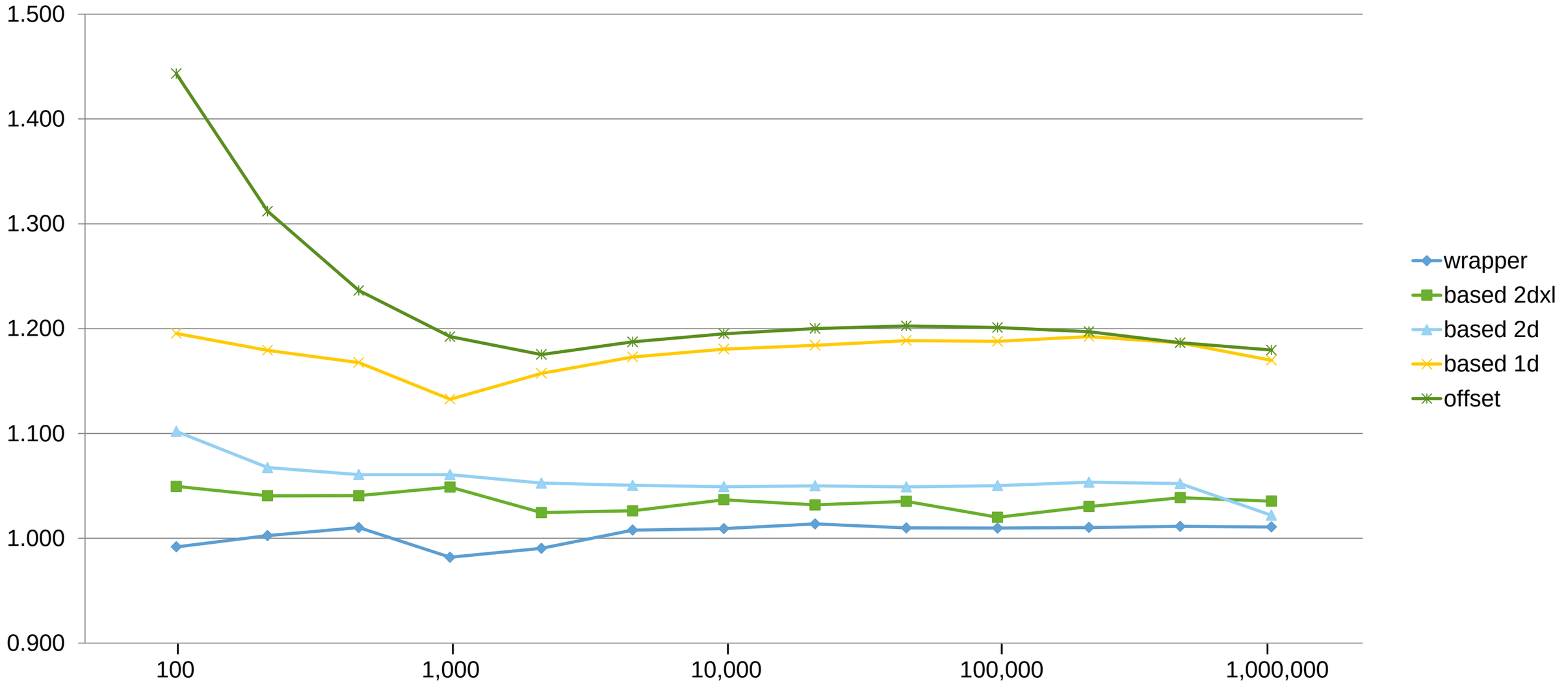
GCC 7.1 / uint64_t / sort()



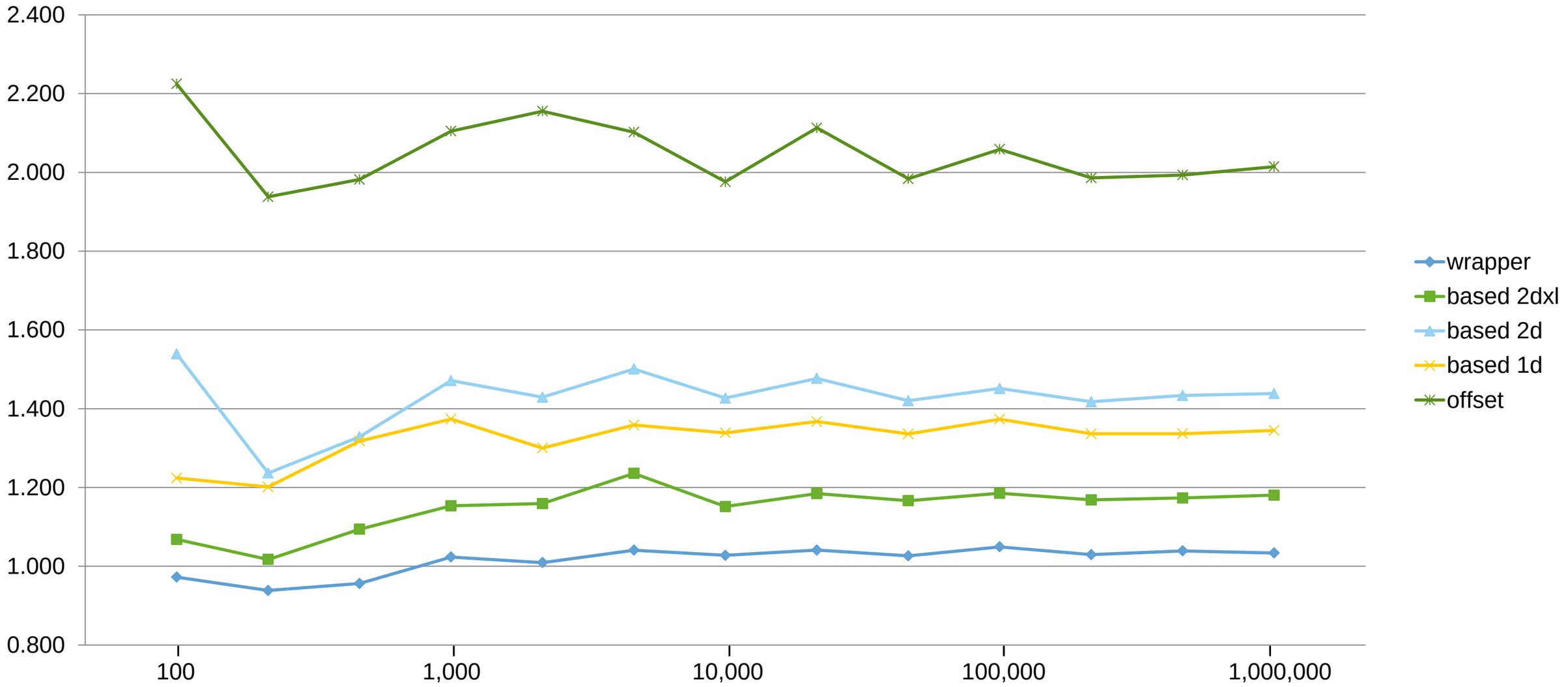
GCC 7.1 / string / sort()



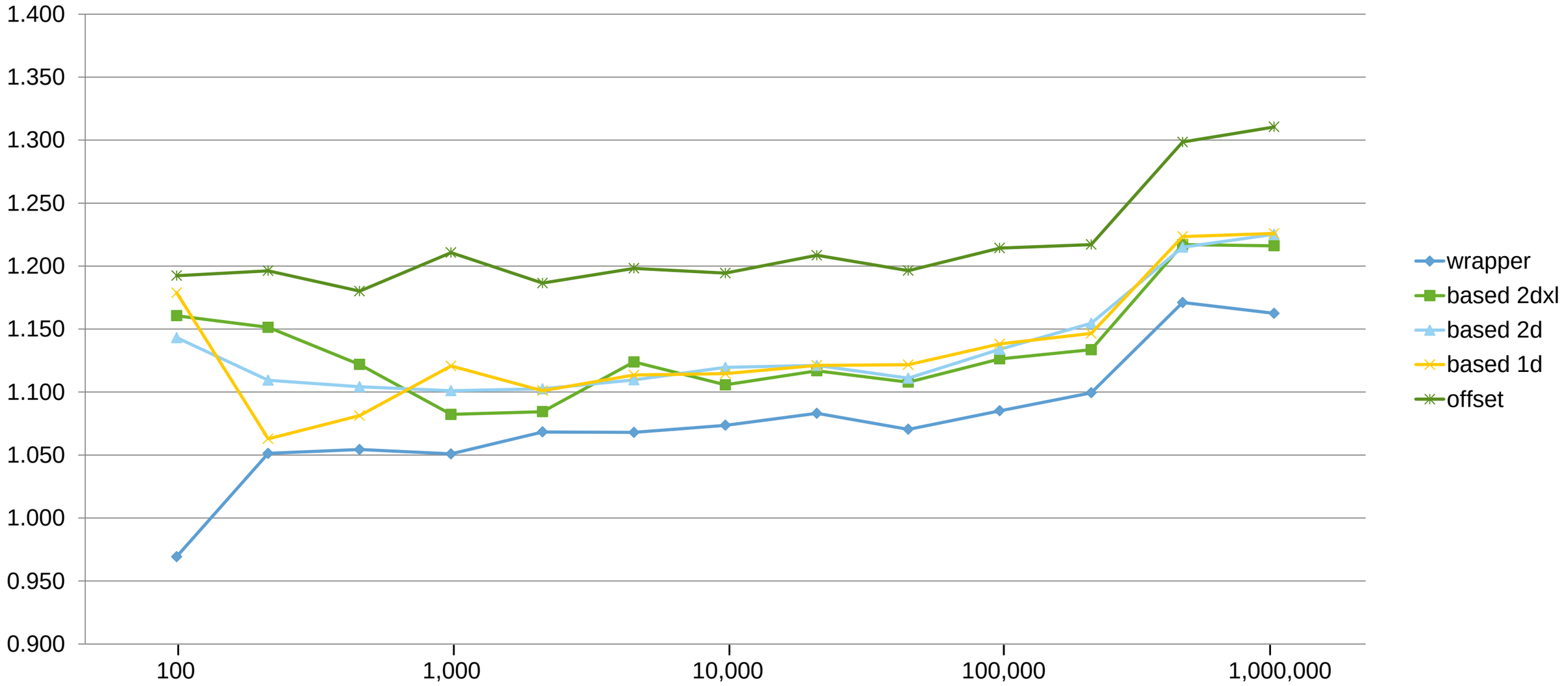
GCC 7.1 / test_struct / sort()



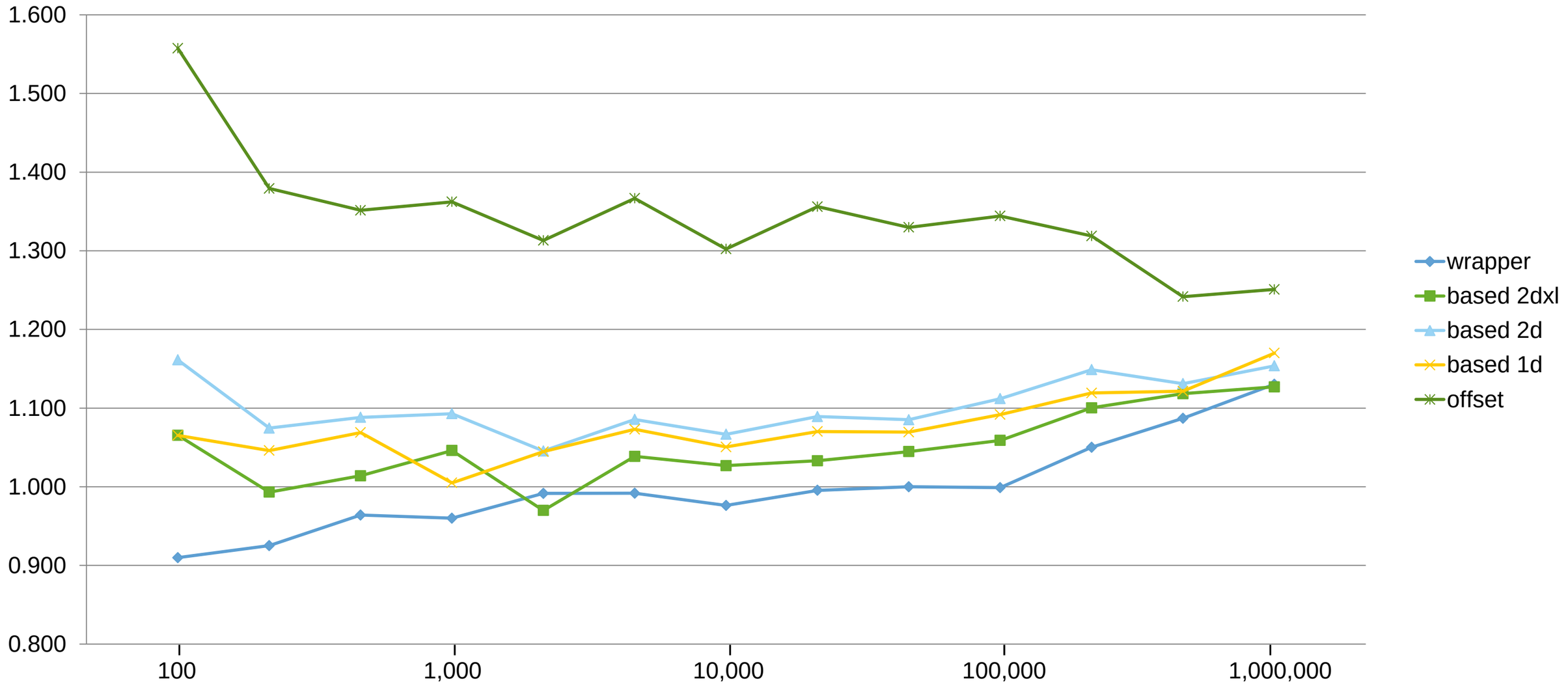
GCC 7.1 / uint64_t / stable_sort()



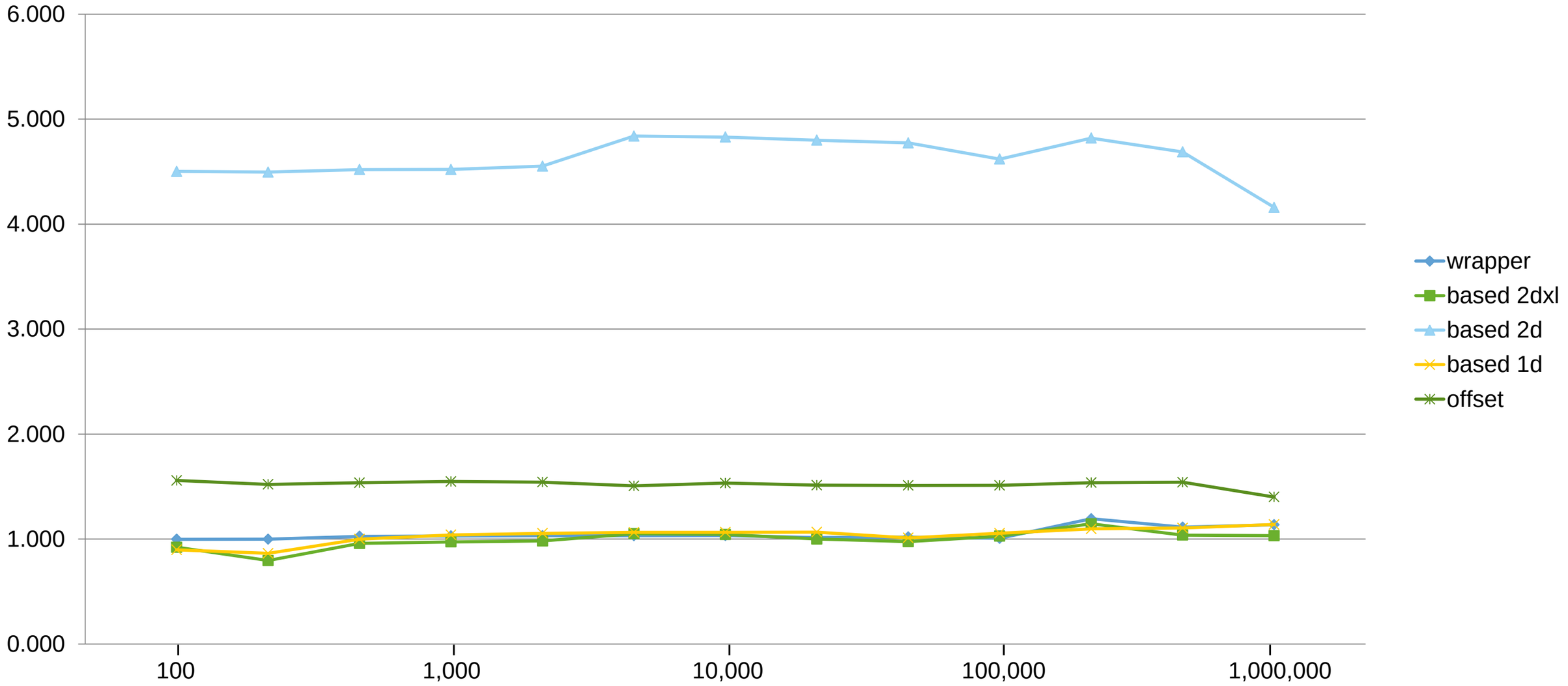
GCC 7.1 / string / stable_sort()



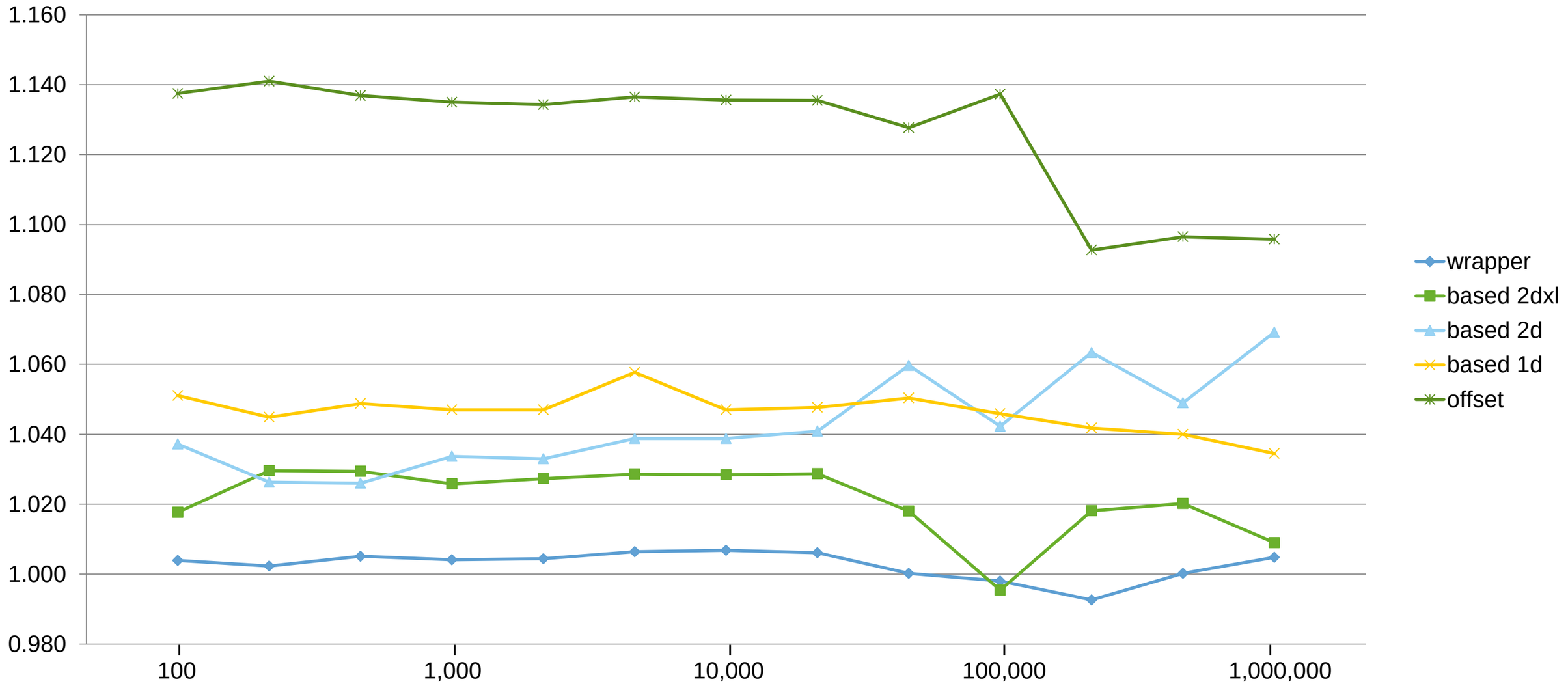
GCC 7.1 / test_struct / stable_sort()



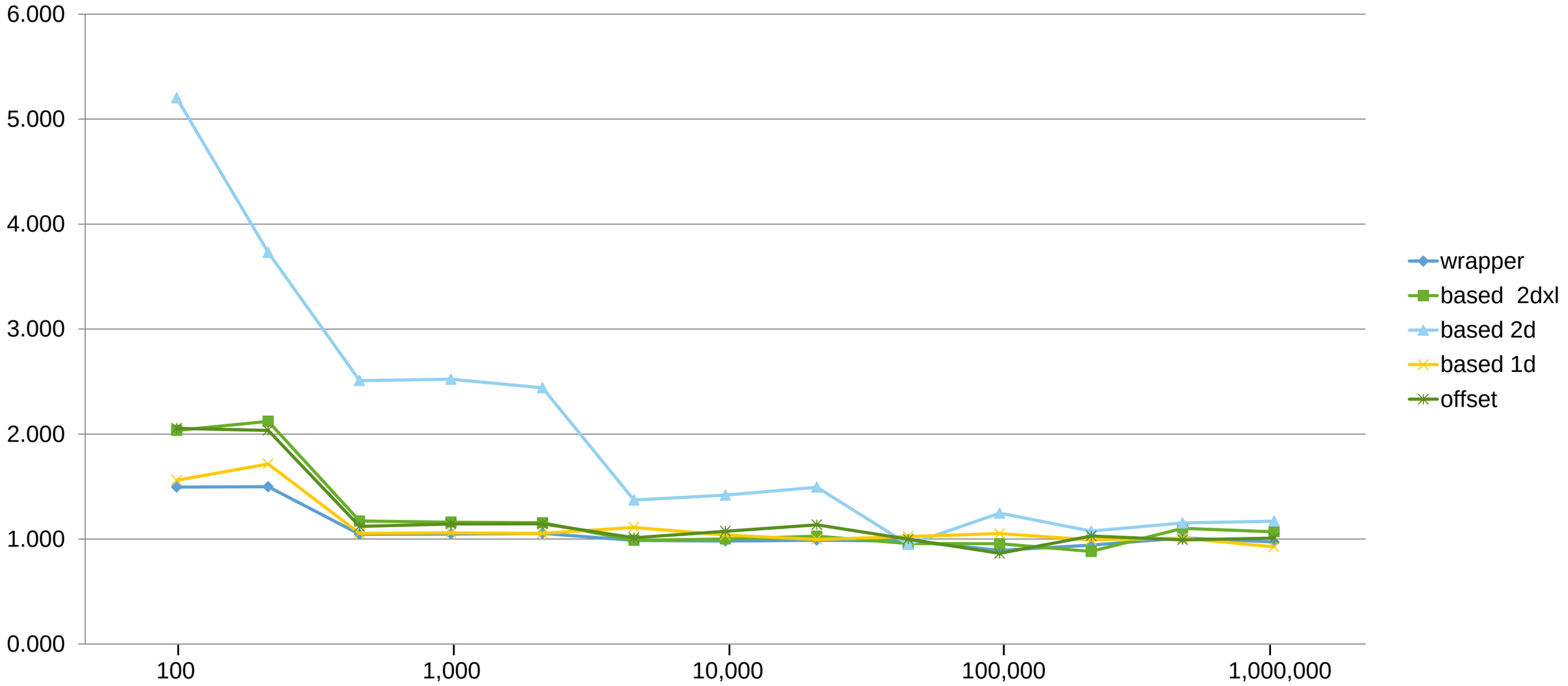
VS 2017 / uint64_t / copy()



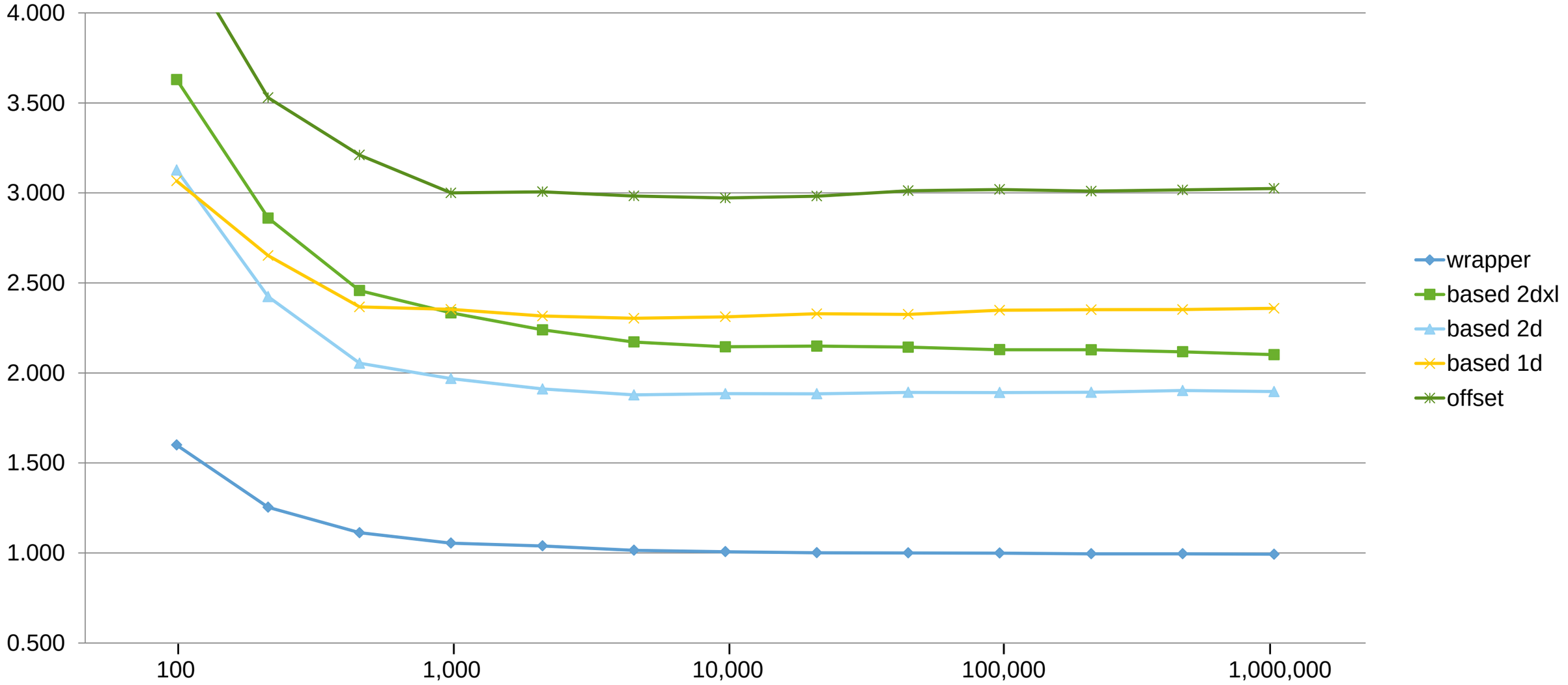
VS 2017 / string / copy()



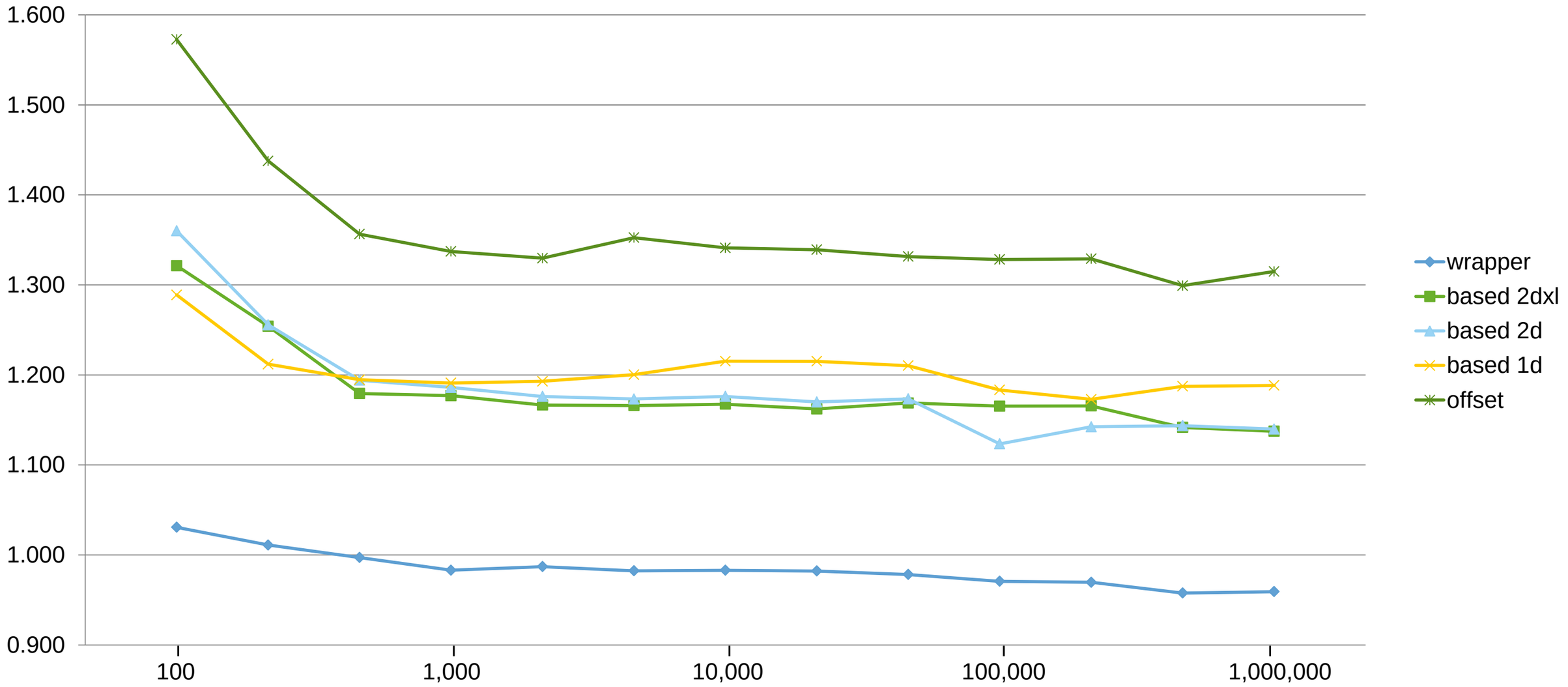
VS 2017 / test_struct / copy()



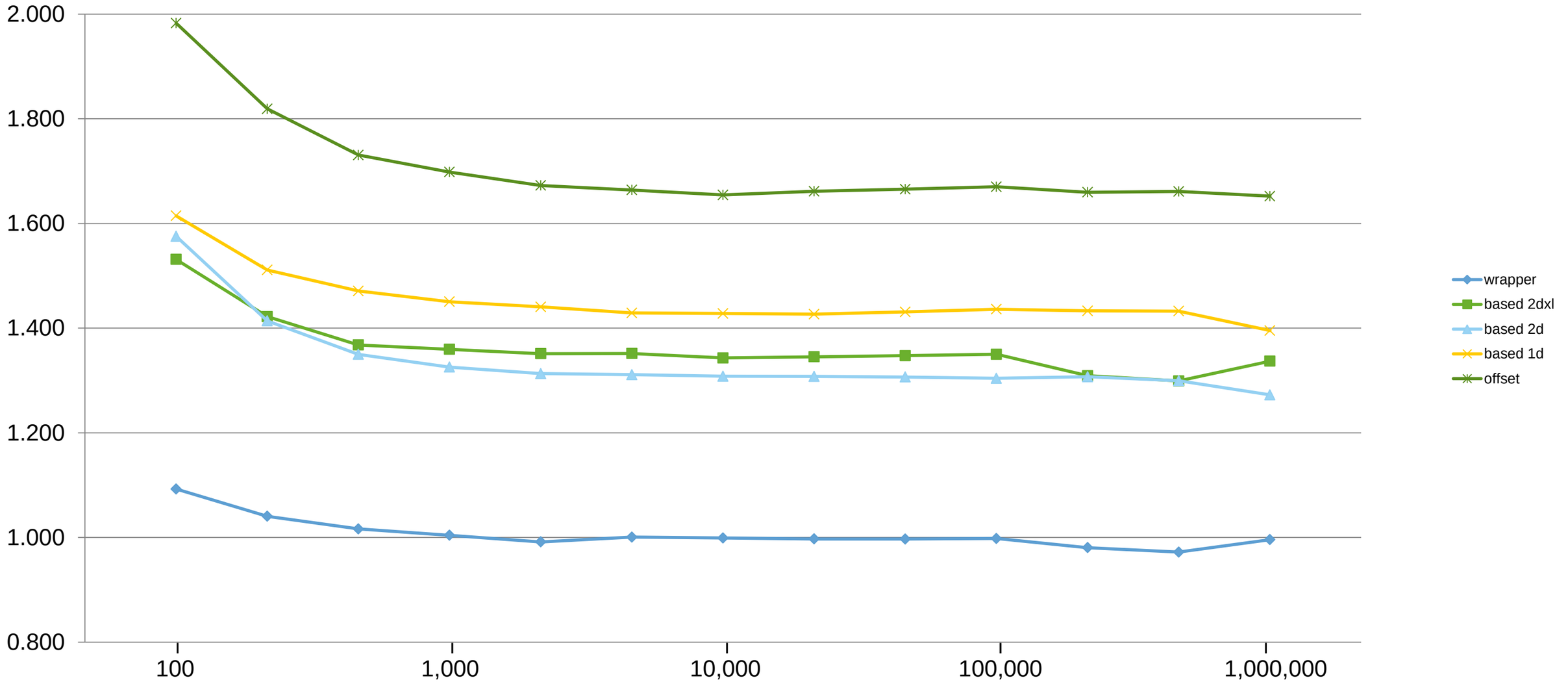
VS 2017 / uint64_t / sort()



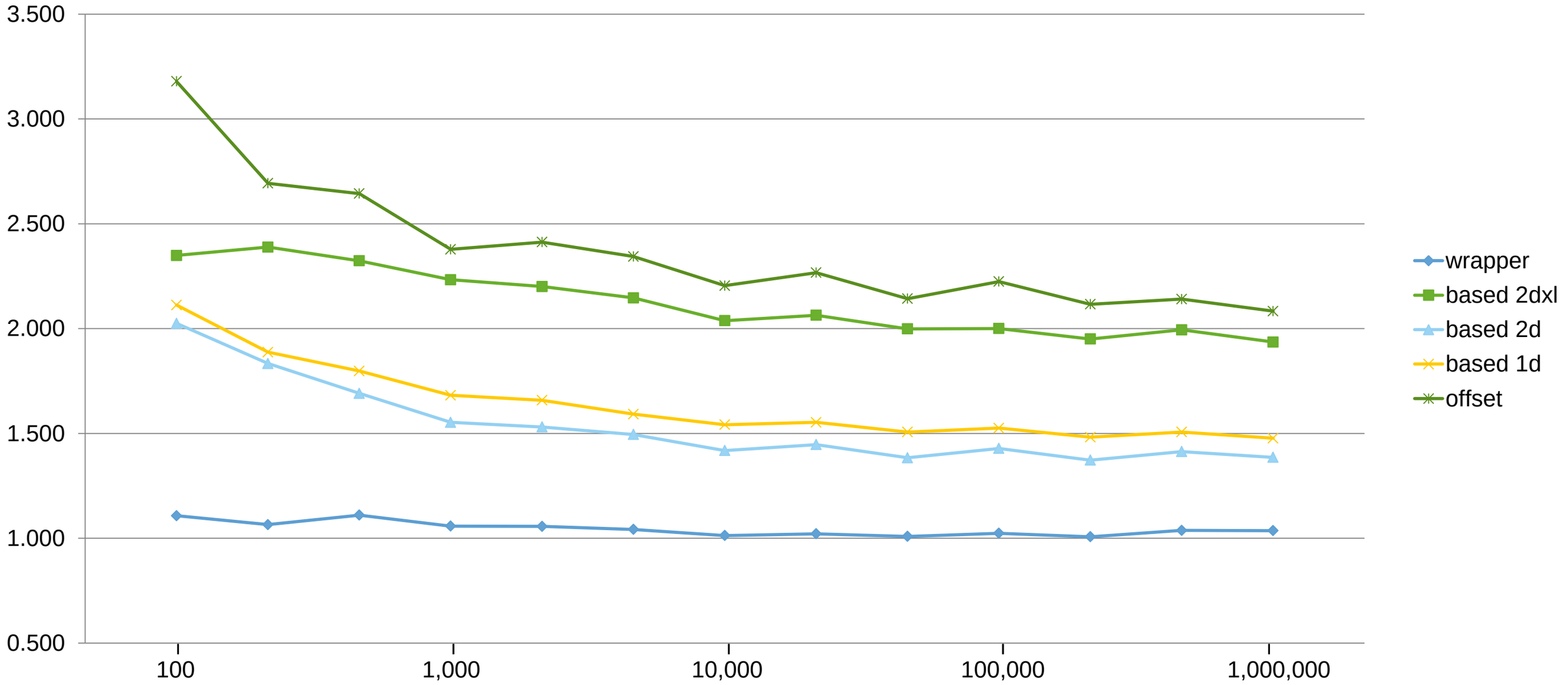
VS 2017 / string / sort()



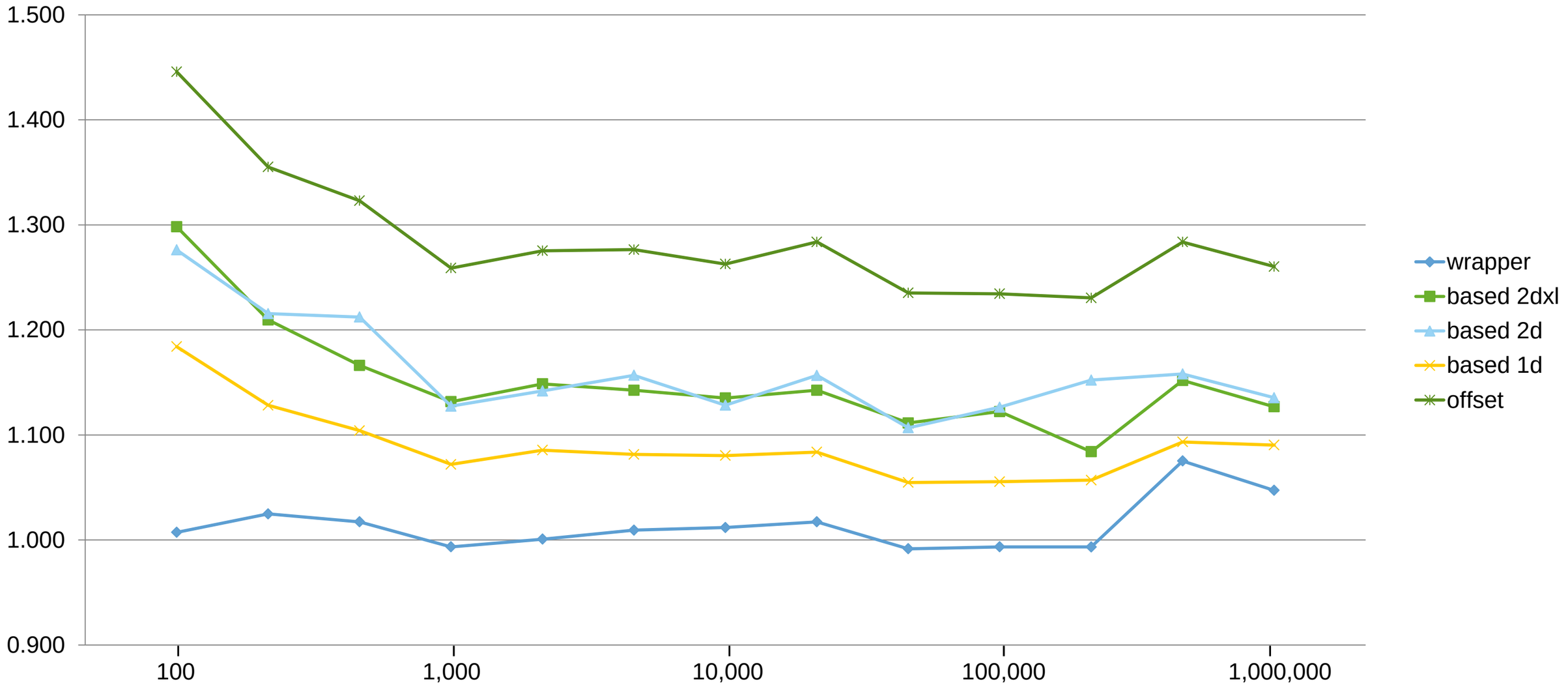
VS 2017 / test_struct / sort()



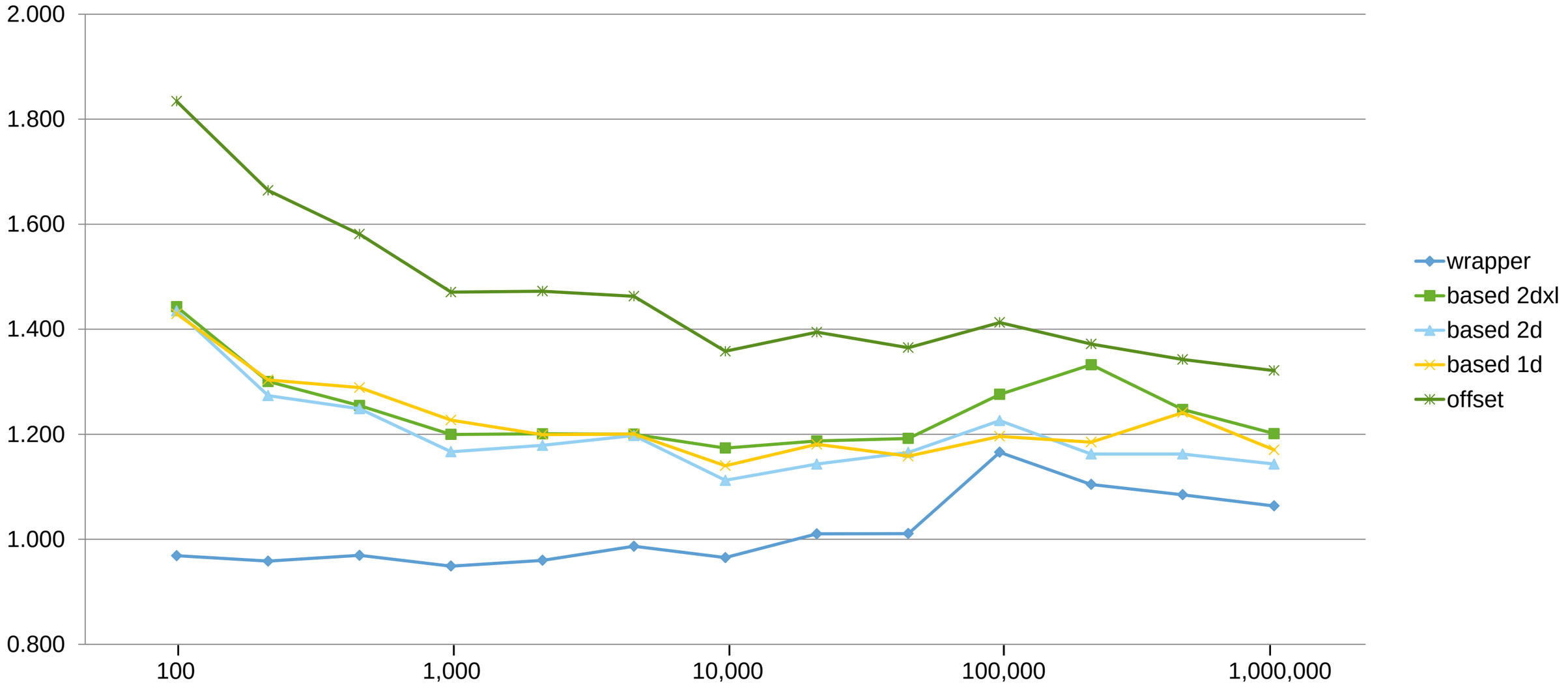
VS 2017 / uint64_t / stable_sort()



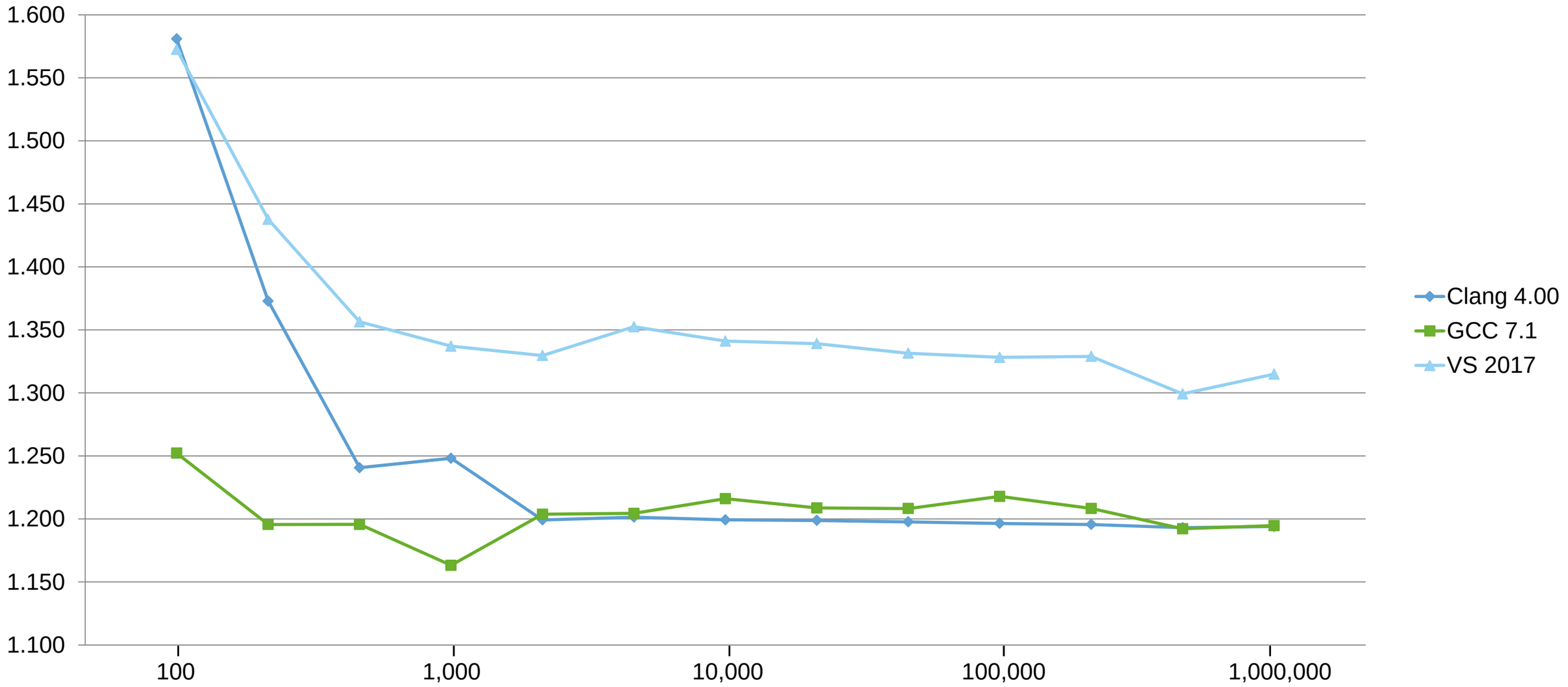
VS 2017 / string / stable_sort()



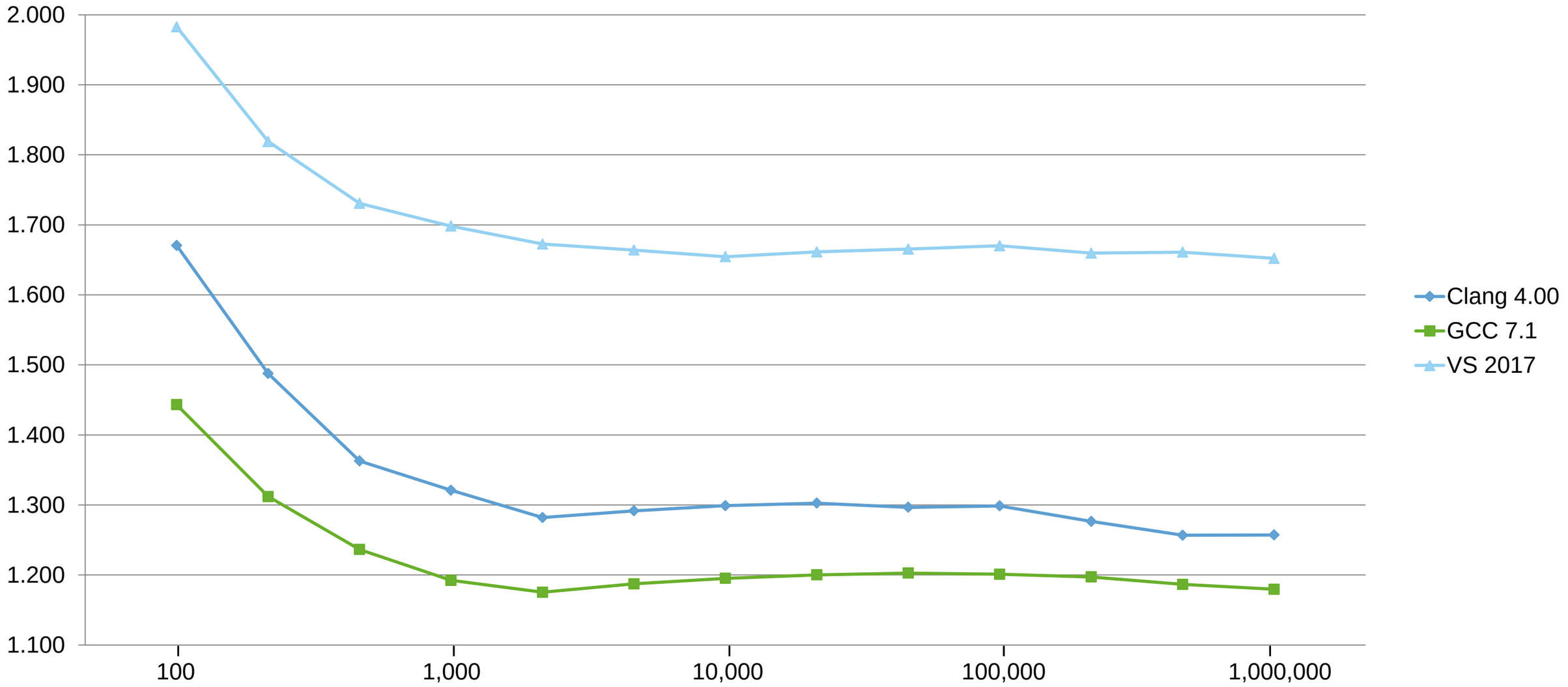
VS 2017 / test_struct / stable_sort()



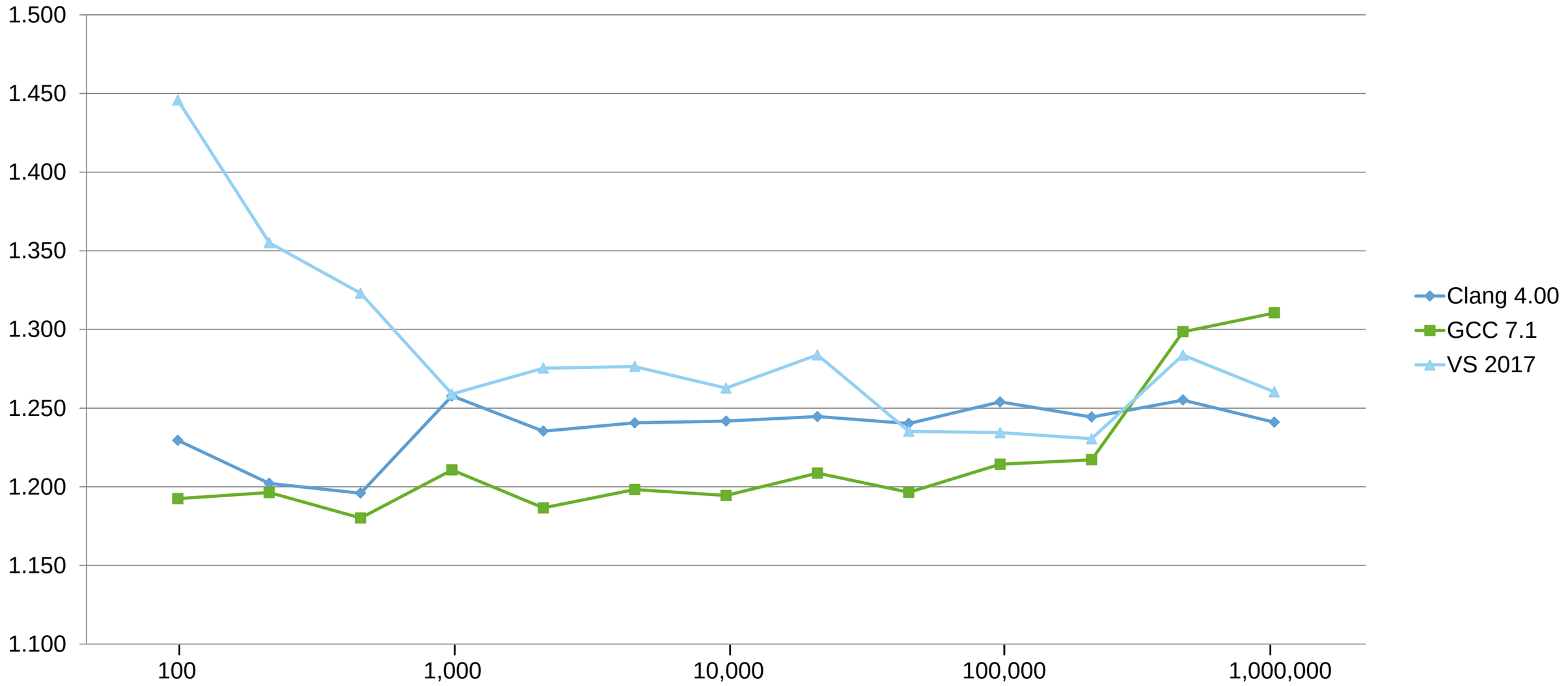
offset_strategy / sort() / string



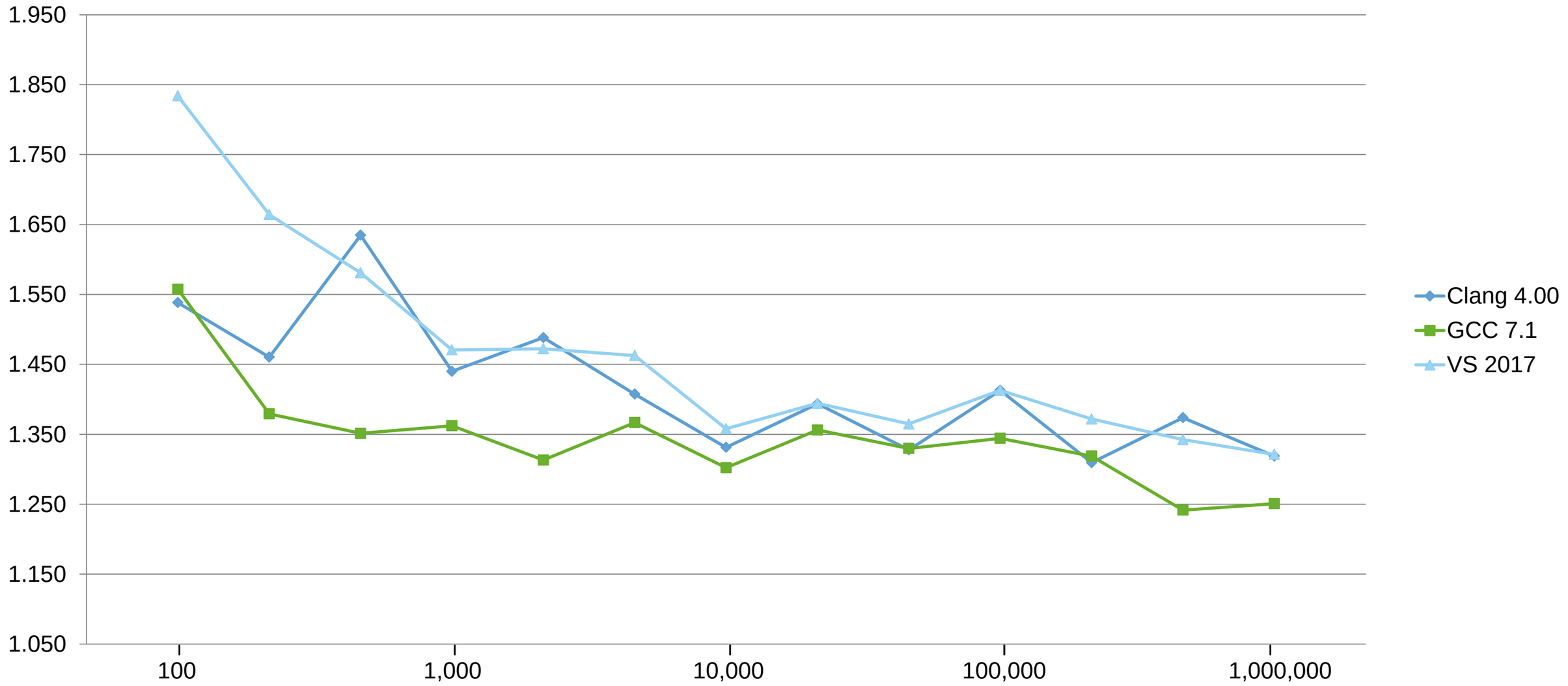
offset_strategy / sort() / test_struct



offset_strategy / stable_sort() / string



offset_strategy / stable_sort() / test_struct



Performance Testing Takeaway

- Offset pointers usually have the worst performance
  - Found anomalies in GCC 5.4/6.3, possible codegen bugs
- It is difficult to predict which addressing model will have the best performance
- There is a surprising variation across data types and across array sizes
- GCC 7.1 usually has the most consistent performance ratios
- GCC 7.1 usually has the performance ratios closest to 1.0

Allocator Awareness Conformance Testing

More Test Framework Types

- Allocator `poc_allocator<T, POCCA, POCMA, POCS, EQ>`
- Allocator `syn_poc_allocator<T, POCCA, POCMA, POCS, EQ>`

pointer \ allocator	stateless	stateful
ordinary	`std::allocator<T>`	`poc_allocator<T,POCCA,POCMA,POCS,EQ>`
synthetic	`rhx_allocator<T,AS>`	`syn_poc_allocator<T,POCCA,POCMA,POCS,EQ>`

Test Allocator poc_allocator

```
template<class T, class POCCA, class POCMA, class POCS, bool EQ>
class poc_allocator
{
  public:
    using propagate_on_container_copy_assignment = POCCA;
    using propagate_on_container_move_assignment = POCMA;
    using propagate_on_container_swap            = POCS;

    using difference_type    = std::ptrdiff_t;
    using size_type          = std::size_t;
    using void_pointer       = void*;
    using const_void_pointer = void const*;
    using pointer            = T*;
    using const_pointer      = T const*;
    using reference          = T&;
    using const_reference    = T const&;
    using value_type         = T;

    ...
};
```

Test Allocator poc_allocator

```
template<class T, class POCCA, class POCMA, class POCS, bool EQ>
class poc_allocator
{
  public:
    ...

    pointer     allocate(size_type n);
    void        deallocate(pointer p, size_type n);

    ...

    std::allocator<T>   m_heap;
};

template<class T, class POCCA, class POCMA, class POCS, bool EQ> bool
operator ==(const poc_allocator<T, POCCA, POCMA, POCS, EQ>&,
            const poc_allocator<T, POCCA, POCMA, POCS, EQ>&);

template<class T, class POCCA, class POCMA, class POCS, bool EQ> bool
operator !=(const poc_allocator<T, POCCA, POCMA, POCS, EQ>&,
            const poc_allocator<T, POCCA, POCMA, POCS, EQ>&);
```

Test Allocator poc_allocator

```
template<class T, class POCCA, class POCMA, class POCS, bool EQ> inline
typename poc_allocator<T, POCCA, POCMA, POCS, EQ>::pointer
poc_allocator<T, POCCA, POCMA, POCS, EQ>::allocate(size_type n)
{
    return m_heap.allocate(n);
}


template<class T, class POCCA, class POCMA, class POCS, bool EQ> inline void
poc_allocator<T, POCCA, POCMA, POCS, EQ>::deallocate(pointer p, size_type n)
{
    m_heap.deallocate(p, n);
}


template<class T, class POCCA, class POCMA, class POCS, bool EQ> inline bool
operator ==(const poc_allocator<T, POCCA, POCMA, POCS, EQ>&,
            const poc_allocator<T, POCCA, POCMA, POCS, EQ>&)
{
    return EQ;    //- and, of course, operator!=() returns !EQ
}
```

Test Allocator syn_poc_allocator

```
template<class T, class POCCA, class POCMA, class POCS, bool EQ>
class syn_poc_allocator
{
  public:
    using propagate_on_container_copy_assignment = POCCA;
    using propagate_on_container_move_assignment = POCMA;
    using propagate_on_container_swap            = POCS;

    using difference_type    = std::ptrdiff_t;
    using size_type          = std::size_t;
    using void_pointer       = syn_ptr<void, wrapper_addressing_model>;
    using const_void_pointer = syn_ptr<void const*, wrapper_addressing_model>;
    using pointer            = syn_ptr<T, wrapper_addressing_model>;
    using const_pointer      = syn_ptr<T const, wrapper_addressing_model>;
    using reference          = T&;
    using const_reference    = T const&;
    using value_type         = T;

    ...
};
```

Test Allocator syn_poc_allocator

```
template<class T, class POCCA, class POCMA, class POCS, bool EQ>
class syn_poc_allocator
{
  public:
    ...

    pointer     allocate(size_type n);
    void        deallocate(pointer p, size_type n);

    ...

    std::allocator<T>   m_heap;
};

template<class T, class POCCA, class POCMA, class POCS, bool EQ> bool
operator ==(const syn_poc_allocator<T, POCCA, POCMA, POCS, EQ>&,
            const syn_poc_allocator<T, POCCA, POCMA, POCS, EQ>&);

template<class T, class POCCA, class POCMA, class POCS, bool EQ> bool
operator !=(const syn_poc_allocator<T, POCCA, POCMA, POCS, EQ>&,
            const syn_poc_allocator<T, POCCA, POCMA, POCS, EQ>&);
```

Test Allocator syn_poc_allocator

```
template<class T, class POCCA, class POCMA, class POCS, bool EQ> inline
typename syn_poc_allocator<T, POCCA, POCMA, POCS, EQ>::pointer
syn_poc_allocator<T, POCCA, POCMA, POCS, EQ>::allocate(size_type n)
{
    return m_heap.allocate(n);
}


template<class T, class POCCA, class POCMA, class POCS, bool EQ> inline void
syn_poc_allocator<T, POCCA, POCMA, POCS, EQ>::deallocate(pointer p, size_type n)
{
    m_heap.deallocate(p, n);
}


template<class T, class POCCA, class POCMA, class POCS, bool EQ> inline bool
operator ==(const syn_poc_allocator<T, POCCA, POCMA, POCS, EQ>&,
            const syn_poc_allocator<T, POCCA, POCMA, POCS, EQ>&)
{
    return EQ;     //- and, of course, operator!=() returns !EQ
}
```

Allocator Awareness Conformance Test Procedure

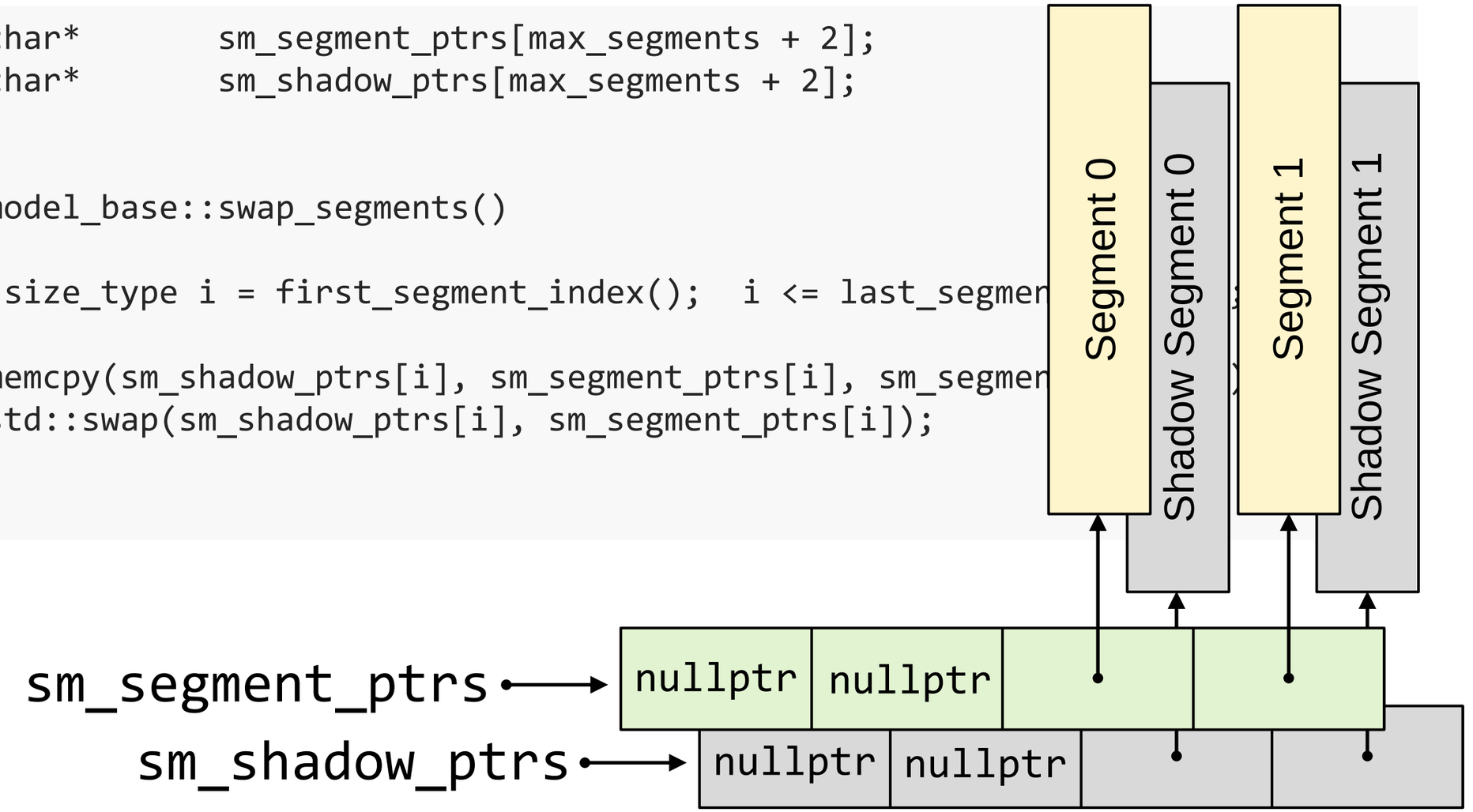
- Compilation and linking succeeds
- Tests run without crashing, hangs, or infinite loops
- Synthetic pointer performance tests yield same results as ordinary pointers
- Container conformance tests using test allocators of `<T>` yield same runtime results as `std::allocator<T>`
- Containers support relocation (relocatable memory segments)
- All tests performed in a single thread
- All results are pass/fail

Container Test - deque

```
template<typename AllocStrategy>
void    run_deque_simple_tests(char const* stype);

template<typename AllocStrategy>
void    run_deque_normal_tests(char const* stype);

template<typename AllocStrategy>
void    run_deque_reloc_tests(char const* stype);

template<class POCCA, class POCMA, class POCS, bool EQ>
void    run_deque_poc_tests();

#define RUN_DEQUE_TESTS(ST)         run_deque_normal_tests<ST>(#ST)

#define RUN_DEQUE_RELOC_TESTS(ST)   run_deque_reloc_tests<ST>(#ST)

#define RUN_DEQUE_POC_TESTS(POCCA, POCMA, POCS, EQ)  \
                                    run_deque_poc_tests<POCCA,POCMA,POCS,EQ>();

void    run_container_deque_tests();
```

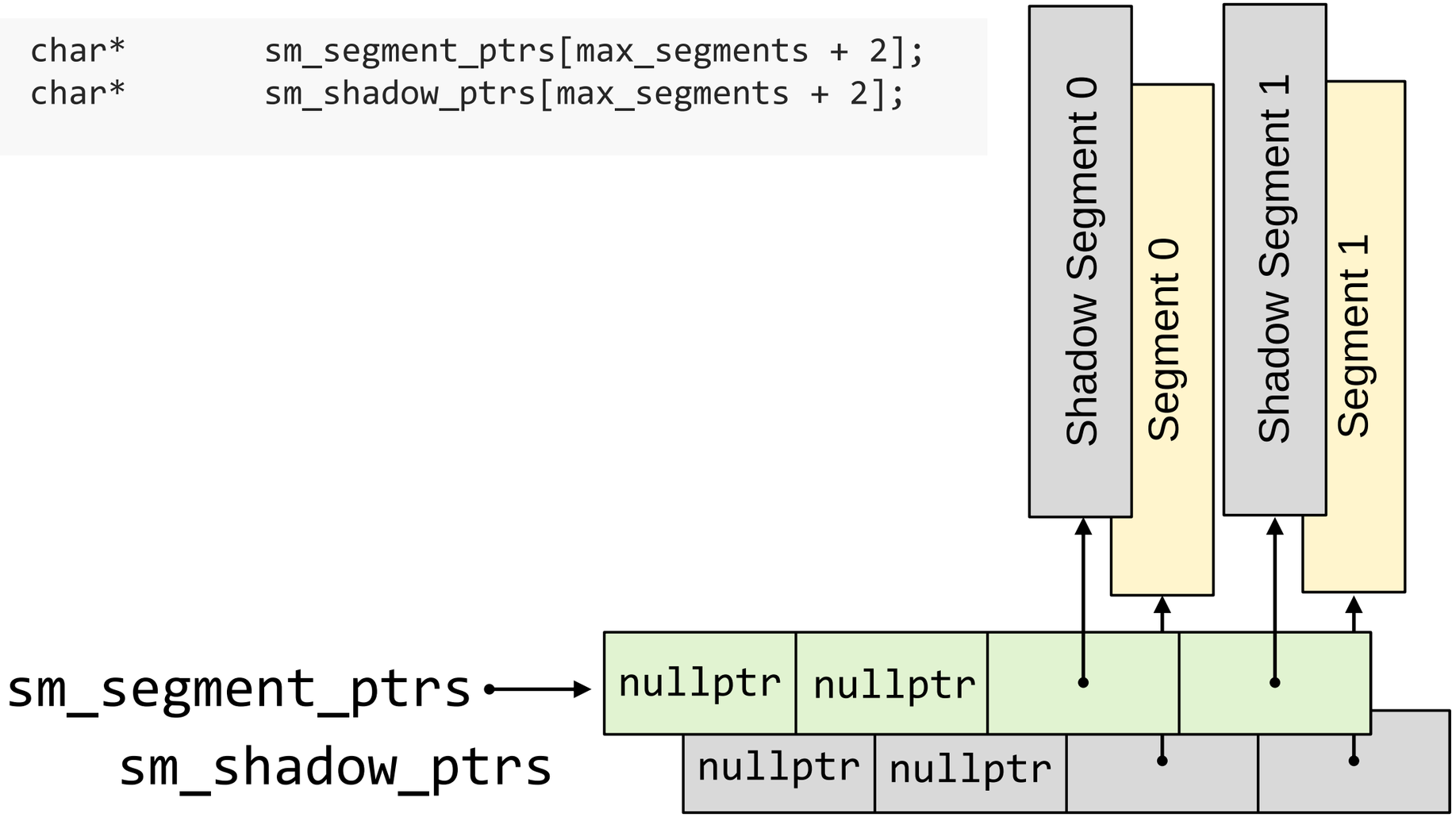
Relocation – storage_model_base::swap_segments()

```
static  char*       sm_segment_ptrs[max_segments + 2];
static  char*       sm_shadow_ptrs[max_segments + 2];

void
storage_model_base::swap_segments()
{
    for (size_type i = first_segment_index();  i <= last_segmen
    {
        memcpy(sm_shadow_ptrs[i], sm_segment_ptrs[i], sm_segmen
        std::swap(sm_shadow_ptrs[i], sm_segment_ptrs[i]);
    }
}
```



Relocation – storage_model_base::swap_segments()

```
static  char*       sm_segment_ptrs[max_segments + 2];
static  char*       sm_shadow_ptrs[max_segments + 2];
```



Container Test - deque

```
#define RUN_DEQUE_TESTS(ST)          run_deque_normal_tests<ST>(#ST)

#define RUN_DEQUE_RELOC_TESTS(ST)    run_deque_reloc_tests<ST>(#ST)

#define RUN_DEQUE_POC_TESTS(POCCA, POCMA, POCS, EQ)  \
                                     run_deque_poc_tests<POCCA,POCMA,POCS,EQ>();

template<typename AllocStrategy>
void    run_deque_normal_tests(char const* stype);

template<typename AllocStrategy>
void    run_deque_reloc_tests(char const* stype);

template<class POCCA, class POCMA, class POCS, bool EQ>
void    run_deque_poc_tests();

void    run_container_deque_tests();
```

Container Test - deque

```
void    run_container_deque_tests()
{
    RUN_DEQUE_TESTS(wrapper_strategy);
    RUN_DEQUE_TESTS(based_2d_strategy);
    RUN_DEQUE_RELOC_TESTS(based_2d_strategy);

    RUN_DEQUE_POC_TESTS(std::true_type,  std::true_type,  std::true_type,  true);
    RUN_DEQUE_POC_TESTS(std::false_type, std::true_type,  std::true_type,  true);
    RUN_DEQUE_POC_TESTS(std::false_type, std::true_type,  std::true_type,  false);

    RUN_DEQUE_POC_TESTS(std::true_type,  std::false_type, std::true_type,  true);
    RUN_DEQUE_POC_TESTS(std::false_type, std::false_type, std::true_type,  true);
    RUN_DEQUE_POC_TESTS(std::false_type, std::false_type, std::true_type,  false);

    RUN_DEQUE_POC_TESTS(std::true_type,  std::true_type,  std::true_type,  true);
    RUN_DEQUE_POC_TESTS(std::true_type,  std::false_type, std::true_type,  true);
    RUN_DEQUE_POC_TESTS(std::true_type,  std::false_type, std::true_type,  false);

    RUN_DEQUE_POC_TESTS(std::false_type, std::true_type,  std::true_type,  true);
    RUN_DEQUE_POC_TESTS(std::false_type, std::false_type, std::true_type,  true);
    RUN_DEQUE_POC_TESTS(std::false_type, std::false_type, std::true_type,  false);
    ...
}
```

Container Test - deque

```
void    run_container_deque_tests()
{
    ...

    RUN_DEQUE_POC_TESTS(std::true_type,  std::true_type,  std::false_type, true);
    RUN_DEQUE_POC_TESTS(std::false_type, std::true_type,  std::false_type, true);

    RUN_DEQUE_POC_TESTS(std::true_type,  std::false_type, std::false_type, true);
    RUN_DEQUE_POC_TESTS(std::false_type, std::false_type, std::false_type, true);

    RUN_DEQUE_POC_TESTS(std::true_type,  std::true_type,  std::false_type, true);
    RUN_DEQUE_POC_TESTS(std::true_type,  std::false_type, std::false_type, true);

    RUN_DEQUE_POC_TESTS(std::false_type, std::true_type,  std::false_type, true);
    RUN_DEQUE_POC_TESTS(std::false_type, std::false_type, std::false_type, true);
}
```

Test Results - GCC 5.4 / 6.3 / 7.1

	Simple/syn	Normal/syn	Reloc/syn	POC/syn	POC/ord
`deque`	Pass	Pass	**Fail**	Pass	Pass
`forward_list`	Pass	Pass	**Fail**	Pass	Pass
`list`	Pass	Pass	**Fail**	**Fail**	**Fail**
`map`	Pass	Pass	**Fail**	Pass	Pass
`string`	Pass	Pass	**Fail**	-	-
`unordered_map`	Pass	Pass	**Fail**	Pass	Pass
`vector`	Pass	Pass	**Fail**	Pass	Pass

- GCC containers convert `allocator::pointer` to `T*` and use `T*` internally
- GCC provides only `hash<T>` specializations; a new specialization had to be created for `basic_string` using the `rhx_allocator`
- GCC POC tests fail when POCS is `true_type` and EQ = `false`
- GCC 5.4 and 6.3 generate bad code in the pointer tests for `uint32_t` and `offset_addressing_model`

Test Results – Clang 3.81 / 3.91 / 4.00

	Simple/syn	Normal/syn	Reloc/syn	POC/syn	POC/ord
`deque`	Pass	Pass	Pass	Pass	Pass
`forward_list`	Pass	Pass	Pass	Pass	Pass
`list*`	Pass	Pass	Pass	Pass	Pass
`map`	Pass	Pass	Pass	Pass	Pass
`string`	Pass**	Pass**	Pass**	-	-
`unordered_map`	Pass	Pass	Pass	Pass	Pass
`vector`	Pass	Pass	Pass	Pass	Pass

- * Clang `list` tests pointers for null by comparing to literal `0`
- ** Clang's `basic_string` implementation does not support the offset addressing model

Test Results – VS 2015 u3

	Simple/syn	Normal/syn	Reloc/syn	POC/syn	POC/ord
`deque`	Pass	Pass	Pass	Pass	Pass
`forward_list`	Pass	Pass	Pass	**Fail**	Pass
`list`	Pass	Pass	Pass	**Fail**	Pass
`map`	Pass	Pass	**Fail**	**Fail**	**Fail**
`string`	Pass	Pass	Pass	-	-
`unordered_map`	Pass	Pass	Pass	**Fail**	Pass
`vector`	Pass	Pass	Pass	Pass	Pass

- POC tests fail to compile
- Relocation test fails at runtime with infinite loop when iterating from `begin()` to end()

Test Results – VS 2017

	Simple/syn	Normal/syn	Reloc/syn	POC/syn	POC/ord
`deque`	Pass	Pass	Pass	Pass	Pass
`forward_list`	Pass	Pass	Pass	Pass	Pass
`list`	Pass	Pass	Pass	Pass	Pass
`map`	Pass	Pass	Pass	Pass	Pass
`string`	Pass	Pass	Pass	-	-
`unordered_map`	Pass	Pass	Pass	Pass	Pass
`vector`	Pass	Pass	Pass	Pass	Pass

- Very impressive support!

Conclusions

- Observations (IMHO)
  - Allocator awareness should be required for `basic_string`
  - Allocator aware containers should be required to use of `allocator::pointer`
- Gold Medal – VS 2017

- A PDF of this talk, and source code will soon be available at
  - https://gitlab.com/BobSteagall/talks/CppNow2017 (that's GitLab, not GitHub)

Thank you for attending!