Università degli Studi di Messina

Dipartimento di Scienze Matematiche e Informatiche,
Scienze Fisiche e Scienze della Terra

Corso di Laurea in Informatica

Analysis of entity encoding techniques, design and implementation of a multithreaded compile-time Entity-Component-System C++14 library

Laureando:
Vittorio Romeo

Relatore:
Prof. Giacomo Fiumara

Anno Accademico 2015/2016

Abstract

Complex real-time games and applications have to deal with enormous amounts of entities that greatly vary in behavior and constantly communicate with each other. Finding an elegant design for entity management that allows developers to quickly build applications from small composable elements, without sacrificing performance, abstraction and safety is a difficult problem - this thesis aims to solve it by designing and implementing a C++14 Entity-Component-System library *ex novo*.

An analyis of common entity encoding techniques is presented, which pinpoints the benefits and drawbacks of approaches such as *object-oriented inheritance*, *object-oriented composition*, and *data-oriented composition*. The **Entity-Component-System** architectural pattern is examined and implemented in **ECST**, a multi-threaded compile-time C++14 library. The design and implementation of the library are explored in detail, using a large number of code snippets and diagrams to aid the readers in appreciating the underlying concepts and architecture.

ECST's objective is to let developers conveniently use the Entity-Component-System pattern at compile-time, striving for maximum performance and a friendly, transparent syntax. Programmers define the high-level program logic in a declarative way, using a dataflow-oriented approach, connecting systems together to generate an implicit dependency directed acyclic graph that is automatically parallelized where possible.

Finally, an example particle simulation implemented using the presented library is analyzed and benchmarked. The components and systems used to encode the particle entities and their relationships are examined. The performance benefits of the multithreading features provided by ECST are measured.

Acknowledgements

I would like to express my utmost appreciation to the following people for their support:

- Prof. **Giacomo Fiumara**, my research supervisor, whose constant involvement and advice were vital to accomplish this paper.
- **Adam Martin**, founder of t-machine.org and of the Entity Systems wiki, who reviewed *Part 1* of this thesis. His articles and his feedback have been extremely valuable.
- **Jackie Kay**, for reviewing a draft version of the complete thesis, providing feedback and constructive criticism.
- **Tom Pollard**, creator of the Markdown template used for this document.
- **Christophe Delord**, developer of pp, who quickly answered my issue reports and feature requests.

In addition, I would like to extend very warm thanks to my collegues whom I've shared three amazing years of study with: *Marco Castano*, *Salvatore Gangemi*, *Francesco Pafumi*, *Giuseppe Attanasio*, *Nazzareno Di Pietro*, *Davide Iuffrida*, *Gianluca Materia*, *Sergio Zavettieri*, *Salvatore Bertoncini*.

Colophon

This thesis was written using the following technologies:

- **Pandoc**, an universal document converter.
  - Most of the document was written in *"Pandoc's extended Markdown"*. Inline LaTeXwas used for pseudocode algorithm blocks *(using the* ***algorithm2e*** *package)* and for the frontispiece.
  - Code snippets are formatted and highlighted thanks to the minted package and the pandoc-minted Pandoc filter.
- **pp**, a "generic document preprocessor with Pandoc in mind". It allows inline inclusion of different diagram types and scripts directly in a Markdown file.
  - **Graphviz** and **PlantUML** were invoked through *pp* to generate most of the graphs present in the thesis.
- **dia**, an open-source diagram editor.

Contents

Chapter 1

Introduction

Successful development of complex real-time applications and games requires a flexible and efficient **entity management** system. As a project becomes more intricate, it's critical to find an elegant way to compose objects in order to prevent code repetition, improve modularity and open up powerful optimization possibilities.

The **Entity-Component-System** architectural pattern was designed to achieve the aforementioned benefits, by **separating data from logic**.

- Entities[1] can be composed of small, reusable, and generic components;
- Components can be stored in contiguous memory areas, thus improving **data locality** and **cache-friendliness**;
- Application logic can be **easily parallelized** and abstracted away from the objects themselves and their storage policies;
- The state of the application can be serialized and shared over the network with less effort;
- A more modular, generic and easily-testable codebase.

The **ECS**[2] pattern will be described and explained in this thesis, alongside an in-depth design and implementation analysis of **ECST**, a **C++14 multithreaded compile-time**[3] **Entity-Component-System** library.

[1] core building blocks of an application.

[2] Entity-Component-System.

[3] *"compile-time"*, in this context, means that component types and system types are known during compilation (i.e. not data-driven).

1.1 Problem/background

Consider a traditional **object-oriented** architecture for a complex application or game: base object classes are defined as roots of huge hierarchies, from which entities derive, containing both data and logic. With the addition of new entity types, the complexity of the code increases, while code reusability, flexibility and performance decrease.

An alternative, more powerful approach consists in using **data-oriented design**[4] *(DOD)* and **composition**[5], where the code is designed around the data and its flow[6], and entities are defined as aggregates of **components**. Data and logic are separated in this approach: auxiliary classes or procedures process data without reasoning in terms of objects, opening up opportunities for functionally pure computations and parallelism. DOD also lends itself to **cache-friendliness**, as storing data separately allows developers to make use of contiguous memory storage.

One common complaint regarding the *DOD + composition* approach is that it seems **less convenient** and **less safe** than an object-oriented approach. While developers praise the flexibility and performance benefits obtained by the aforementioned techniques, those praises are often accompanied by the idea that **abstraction** and **encapsulation** have to be sacrificed in order to get the benefits those techniques aim to achieve.

1.1.1 Objectives

- Objectively analyze **object-oriented design** techniques versus **data-oriented design** + **composition**.
  - A *by-example* examination of a gradual approach shift (from object-oriented inheritance to data-oriented composition) will be presented in Part 1.
- Analyze the design, architecture and implementation of **ECST**, a C++14 compile-time Entity-Component-System library, developed as the thesis

[4] development approach where "it's all about the data", which drives the programmer to design the code around the data. Can provide significant performance and reusability benefits.

[5] avoidance of polymorphic hierarchies in favor of multiple reusable small pieces of data and/or logic that, when put together, form entities.

[6] set of chained data transformations that may depend on each other.

project, in Part 2.

  - Prove that it's not necessary to sacrifice typical OOP advantages (like encapsulation or code reusability), thanks to C++14 **cost-free abstractions** that make use of compile-time knowledge to increase productivity, safety, maintainability and code quality.

- Examine the design and implementation of a simple **real-time particle simulation** developed using ECST in Part 3.

  - Various combination multithreading compile-time options will be compared through benchmarks.

1.2 Related literature

Literature on the Entity-Component-System pattern is limited and hard to find. On the contrary, entity management, especially in the context of game development, has been extensively covered: see [1, Ch. 14], [2, Sec. 1.8], [3, Sec. 1.3], [4, Sec. 4.6], and [5]. Heavily composition-based approaches can be found in [6] and in [7]

Countless online articles and blog posts on the ECS pattern and on data-oriented design have been written - AAA projects postmortems, presentations, well-documented libraries and other kind of valuable resources can be easily found. An excellent introduction to composition can be found in [8]. One of the most comprehensive sets of articles on entity systems was written by Adam Martin, and can be found in [9].

1.3 Code

ECST's source code can be found in the following GitHub repository under the *Academic Free License ("AFL") v. 3.0*: https://github.com/SuperV1234/ecst. The source code for the thesis and for the example particle simulation can be found in the following GitHub repository under the *Academic Free License ("AFL") v. 3.0*: https://github.com/SuperV1234/bcs_thesis.

Note to readers: most of the code snippets included in the thesis are simplified in order to make them more easily understandable and less cluttered. Features like `noexcept` and boilerplate code like repeated *ref-qualified* member functions are

intentionally not included. From [Part 2] onwards, the reader is expected to be familiar with advanced C++11 and C++14 features.

1.4 Long-term research

Research on the Entity-Component-System pattern and its possible implementations has been a long-term personal project for the thesis author:

- SSVEntitySystem, a naive implementation of a single-threaded ECS making use of run-time polymorphism, was released as an open-source library in 2012;
- A singlethreaded compile-time Entity-Component-System implementation tutorial talk was presented at CppCon 2015 *(Bellevue, WA)*. All material used during the talk is available in the following GitHub repository: https://github.com/SuperV1234/cppcon2015;
- Development of ECST started in December 2015. An earlier version of the library was presented in May 2016 at C++Now 2016 *(Aspen, CO)*. The material used during the presentation is available in the following GitHub repository: https://github.com/SuperV1234/cppnow2016.

Part I

The Entity-Component-System pattern

Chapter 2

Overview

Entity-component-system *(ECS)* is a software development architectural pattern suited for complex applications and games that benefit from defining objects in terms of smaller, reusable parts. The ECS pattern embraces the **"composition over inheritance"** principle, solving the issues caused by deep inheritance hierarchies and introducing significant **performance**, **flexibility** and **productivity** advantages.

The pattern consists of three elements that interact with each other:

- **Entities**: defined by Adam Martin in [10] as "fundamental conceptual building blocks" of a system, which represent concrete application objects. They have no application-specific data or logic.
- **Components**: small, reusable, types that compose entities. Again, citing Adam Martin in [10], a component type "labels an entity as possessing a particular aspect". Components store data but do not contain any logic.
- **Systems**[1]: providers of implementation logic for entities possessing a specific set of component types.

In this chapter, the history and some use-cases of the Entity-Component-System pattern will be briefly explored. Afterwards, in Chapter 3, a gradual transition of **entity encoding** techniques, from *"traditional" object-oriented inheritance* to a *data-oriented*[2] approach, will be shown and analyzed.

[1] a.k.a. **processors**.

[2] **data-oriented** design is a development technique that primarily focuses on data *(instead of*

2.1 History and use cases

The Entity-Component-System pattern is heavily used today, especially in AAA game development.

One of the earliest uses of a composition-based architectural pattern in AAA game development can be found in **Thief: The Dark Project (1998)** [11], where the usage of a data-driven system focused on the creation of small reusable game engine components was considered a "risk" that paid off in terms of productivity and quality of the end product.

A similar data-driven component-based approach was implemented in **Dungeon Siege (2002)** - Scott Bilas, one of the engine developers, explained the techniques used in **A Data-Driven Game Object System** at GDC San Jose 2002 [12].

Many similar lectures or project postmortems can be found, praising the benefits of composition and the additional productivity provided by data-driven development. Of notable interest, **Evolve Your Hierarchy** [13], by Mick West, is an article written in 2007 that clearly shows the advantages of composition over inheritance, and was a well-received introduction to the ECS pattern for many game developers.

Another excellent example can be found in Terrance Cohen *(Insomniac Games)*'s **A Dynamic Component Architecture for High Performance Gameplay** GDC Canada 2010 lecture [14].

While academic papers and publications on the subject are rare to find, there currently are countless reports and case studies of successful uses of the ECS pattern in games and applications, ranging from VFX and computer graphics [15], to independent *roguelike* game development [16].

objects), separated from logic. The code is designed around the data, potentially resulting in a more efficient *(due to cache-locality, easier parallelization opportunities and lack of polymorphism overhead)* implementation. Data-oriented design is orthogonal to **data-driven** programming, which is a paradigm where the flow of the program is controlled by external data at run-time.

Chapter 3

Encoding entities

The *essence* of the problem that ECS solves is finding an efficient and flexible way of **encoding entities**.

After trying to define the term *"entity"*, multiple entity encoding techniques will be covered in this chapter, starting from **object-oriented inheritance** and gradually moving to **data-oriented composition**. To fairly compare the approaches, two example application designs will be illustrated beforehand, and then implemented with every analyzed technique.

3.1 Definition of entity

Finding a formal and universal definition for the term *"entity"* in the context of application and game development is not an easy task. Nevertheless, a number of properties closely related to the *idea of entity* can be listed:

- An entity is something closely related to a specific concept;
- Entities have related data and related logic;
- Entities are stored, managed and processed in large quantities;
  - In addition, some particular entity instances may need to be tracked.
- Entities can be created and destroyed during program execution.

Theoretically speaking, entities could be thought of as the **fundamental building blocks** of an application. Following this line of reasoning, pattern-specific elements like components and systems, polymorphism and inheritance, or scripts and definitions in data-driven architectures may be simply considered *implementation details*.

Examples of entities include:

- **Game objects**, like **projectiles**, **walls**, and **power-ups**;
- **Widgets** in a GUI framework, like **buttons** and **textboxes**;
- **Client states** in a server;
- **Particles** and **lights** in VFX creation software.

It is also important to distinguish between **entity types**[1] and **entity instances**. The former is the set of properties, behaviors, aspects and ideas that all instances of a particular entity type share - like a *blueprint*. The latter refers to single instantiations of particular entity types, that can be created, tracked, destroyed, inspected and modified.

[1] the word *"type"* does not necessarily refer to types in programming languages - many engines with typeless entities *(data-driven)* do exist. The intended meaning is "class of entity instances with same components that model the same concept".

3.2 Example use cases

To compare the aforementioned entity encoding approaches as fairly as possible, two extremely simple hypothetical application designs will be provided in this section. A possible implementation of both designs will be shown in the following sections, in order to provide readers with example code and diagrams roughly resembling real-world applications.

Minimal designs for a fantasy **role-playing game** and a bare-bones **GUI framework** are described in the subsections below.

3.2.1 Role-playing game

The designed imaginary RPG will have the following **NPCs** *(non-playing characters)*:

- A **warrior**, that can be unarmed or wield any combination of **sword** and **bow**;
- A **skeleton**;
- A flying **dragon**.

Giving the warrior the possibility of wielding a combination of weapons *(or of being unarmed)* poses interesting design decisions when creating object-oriented class hierarchies.

3.2.2 GUI framework

This hypothetical GUI framework will have the following **widget** types:

- A **textbox**, which supports keyboard input;
- A **button**, which supports mouse input.

Both widget types can be **optionally animated**.

Allowing optional widget animations involves finding an optimal way of avoiding repetition or unused data/logic during entity encoding.

3.3 Object-oriented inheritance

A reasonably easy way of encoding entities consists in using a **hierarchy of polymorphic objects** - this technique *feels natural* to developers accustomed to OOP principles. The concepts of *"component"* and *"system"* will not appear in this approach: both **data** and **logic** are stored inside entities. Entity types encoded using this technique produce hierarchy graphs similar to the following one:

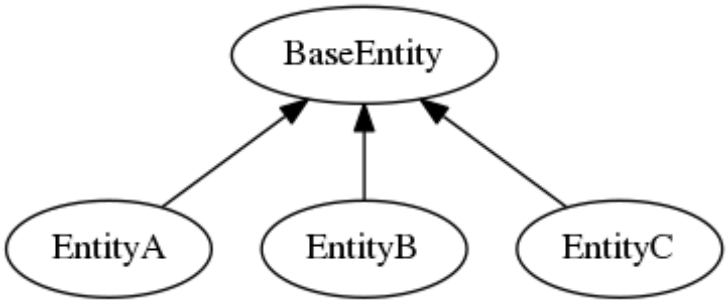


Figure 3.1: Object-oriented inheritance: hypothetical entity hierarchy

3.3.1 Implementation

This approach does not require any technique-specific implementation detail. Since entity types have to conform to the same interface *(due to the use of run-time polymorphism)*, the *"base entity"* class is defined directly in the application's codebase. The provided `virtual` interface greatly depends on the application itself. Some sort of *"manager"* class is usually defined as well, which keeps track of the active entities and provides a convenient interface to perform actions on them.

3.3.1.1 Role-playing game

To simulate the implementation of the example RPG, a hypothetical class hierarchy is shown below:

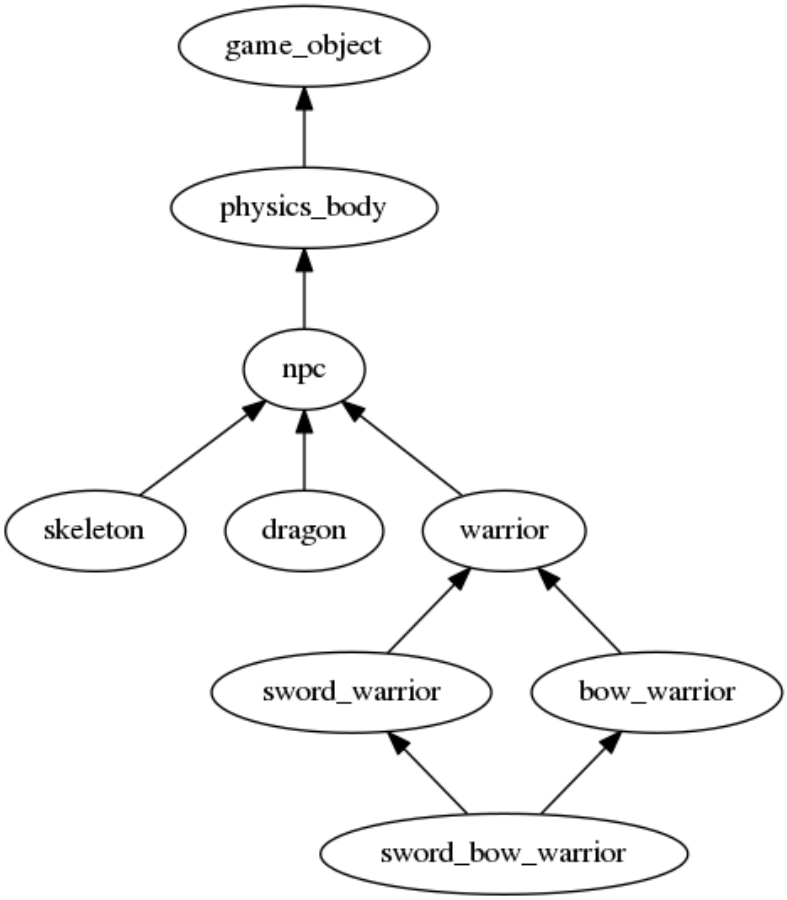


Figure 3.2: Object-oriented inheritance: RPG entity hierarchy

The `game_object` entity type will be the base class from which all other game objects derive - the interface has to be defined there:

```
1  class game_object
2  {
3  public:
4      virtual ~game_object() { }
5
6      virtual void update(float dt) = 0;
7      virtual void draw() = 0;
8  };
```

The rest of the hierarchy is built upon `game_object`, **incrementally** adding data and logic. Every game object that follows the laws of physics will inherit from `physics_body`:

```
class physics_body : game_object
{
private:
    vector2f _position, _velocity;

public:
    void update(float dt) override
    {
        _position += _velocity * dt;
    }
};
```

Physical objects that can be rendered and controlled by an AI will inherit from `npc`:

```
class npc : physics_body
{
private:
    model* _model;
    texture* _texture;
    ai* _ai;

public:
    void update(float dt) override
    {
        physics_body::update(dt);
        _ai->think(dt);
    }

    void draw() override { /* ... */ }
};
```

The leaves of the hierarchy tree will contain highly-specific data and logic:

```
class skeleton : npc { /* ... */ };
class dragon : npc { /* ... */ };
class warrior : npc { /* ... */ };

class sword_warrior : virtual warrior { /* ... */ };
class bow_warrior : virtual warrior { /* ... */ };

class sword_bow_warrior : sword_warrior, bow_warrior
{
    /* ... */
};
```

To avoid code repetition during the implementation of a warrior that simultaneously uses both a sword and a bow, a situation that requires *multiple inheritance* arises. A possible way of avoiding duplicating the contents of the `warrior` class twice in `sword_bow_warrior` is using C++'s `virtual` inheritance feature, concisely explained

in [17]. These kind of hierarchies making use of multiple inheritance are extremely cumbersome to maintain and expand: the pattern appearing in `sword_bow_warrior` is in fact commonly called *"diamond of death"*.

3.3.1.1.1 "Diamond of death" The **"diamond of death"** *(a.k.a. "deadly diamond" or "common ancestor" issue)* is a well-known problem that can appear in object-oriented hierarchies. It was comprehensively covered in [18]. An example is found in this part of the previous hierarchy:

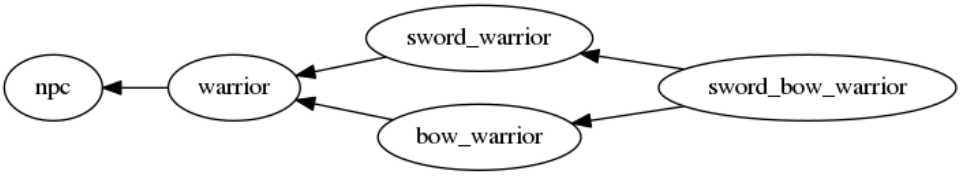


Figure 3.3: OOP encoding issue: "diamond of death"

The hierarchy causes **ambiguity** in case both `sword_warrior` and `bow_warrior` override the same method in `warrior` and causes **duplication** of `warrior`'s fields in `sword_bow_warrior`. To solve these problems, unless a feature like C++'s `virtual` inheritance is used, the class hierarchy has to be altered to avoid the "diamond", potentially introducing code repetition.

3.3.1.2 GUI framework

An example implementation of the previously described imaginary GUI framework will now be covered. As with the role-playing game, the proper starting point is the class hierarchy. The base class of the framework will be called `widget`. Unfortunately, the feature that allows widgets to be optionally animated impedes the creation of a straightforward diagram and introduces the **repetition problem**.

3.3.1.2.1 Repetition problem Handling optional animation features is problematic: a possible approach would be defining a base `animated_widget` class and have all widgets derive from it.

Figure 3.4: OOP encoding issue: repetition #0

In the case shown above, all instances of `textbox` and `button` will have potentially unnecessary fields and methods if they do not make use of the framework's animation capabilities. To solve this, one may try to restructure the hierarchy as follows:

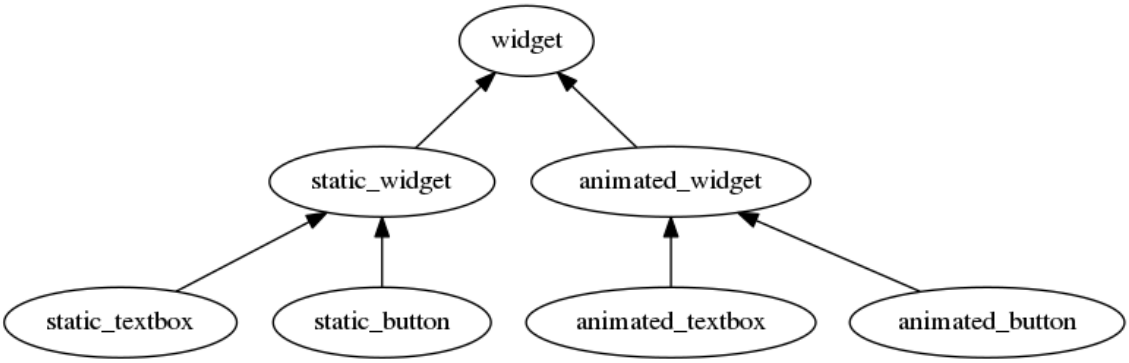


Figure 3.5: OOP encoding issue: repetition #1

Or in the following way:

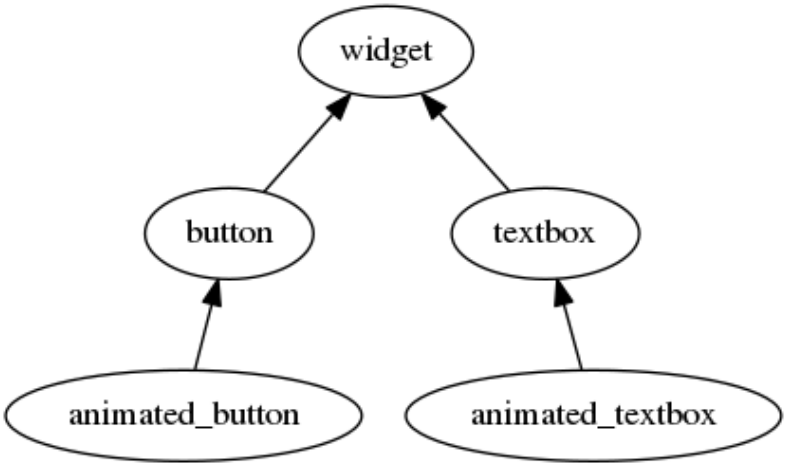


Figure 3.6: OOP encoding issue: repetition #2

As the graphs show, **there is always some form of repetition in one way or**

another:

- In the first hierarchy, the widget data/logic for `textbox` and `button` has to be repeated for the `static_widget` and `animated_widget` branches of the hierarchies.
- In the second hierarchy, the animation data/logic has to be repeated both for `textbox` and `button`.

Upon deciding between one of the hierarchies, the implementation does not differ much from the RPG one: a `widget` base class will be created, with an interface all derived widgets must conform to. The derived widget types will override `widget`'s methods to implement their logic.

3.3.2 Communication

Entity communication in this approach may need to occur for several reasons, including:

- One entity needs to notify another entity about a particular event.
  - *Example:* the close button entity needs to inform the parent window that it needs to be closed.
  - *Example:* a ball hitting a brick in a Breakout clone needs to destroy the brick.
- An entity may be an aggregate of entities that need to cooperate.
  - *Example:* a `tank` entity may be "composed" by a `cannon` entity and a `tracks` entity.

3.3.2.1 Address-based

Since entities are dynamically allocated, their location in memory does not change during program execution. Senders that are **aware of the lifetimes of their listeners** can effectively store pointers *(or references)* to them, in order to directly execute some of their methods or mutate their `public` fields.

```
class window : widget { /* ... */ };

class close_button : widget
{
private:
    window& _parent;

    // ...initialize fields on construction...

    void click()
    {
        _parent.close();
    }
};
```

If senders are not aware of how long their receivers will live, or if senders need to conditionally execute code depending on whether or not a specific receiver is alive, this approach fails. Additional drawbacks include harder networking and serialization, tightly coupled code which is hard to reason about, and lack of flexibility. A possible alternative given by the introduction of a *mediator* class that deals with entity creation and destruction, keeping track of entity states.

The mediator could generate **handles** that, similarly to pointers, would allow access to entities' memory locations, but also add an additional layer of indirection where the validity of the entity pointed to by the handle can be checked.

3.3.2.2 Subscription-based

Using a pattern similar to **"signals and slots"** or **"events and delegates"**, it is possible to implement inter-entity communication by subscribing/unsubscribing to events and implementing event handlers. This approach solves the problem where the sender is not aware of the lifetime of the receiver - the onus of subscribing to specific event types lies on the receiver, that can unsubscribe itself upon destruction.

The `game_object` class will contain an `on_destruction` event that will be invoked on entity destruction:

```
class game_object
{
public:
    event on_destruction;
    // ...
};
```

Game objects that need to communicate, like `tank`, will provide subscribable events:

```
class cannon : game_object { /* ... */ };
class tracks : game_object { /* ... */ };

class tank : game_object
{
public:
    event on_fire;
    // ...
};
```

Entities will be *"linked"* together after they have been created:

```
void link_tank_events(tank& tk, cannon& cn, tracks& ts)
{
    auto event_handle = tk.on_fire.subscribe([&]
        {
            ts.stop();
            cn.fire();
        });

    auto unsub_on_fire = [&tk, event_handle]
        {
            tk.on_fire.unsubscribe(event_handle);
        };

    cn.on_destruction(unsub_on_fire);
    ts.on_destruction(unsub_on_fire);
}
```

While this approach provides some decoupling between entity types, it also introduces unnecessary run-time overhead due to event management. It is also unsafe, as application code needs to make sure entities with subscribers outlive their subscribers *(e.g. `tk` needs to outlive both `cn` and `ts`)*.

3.3.2.3 Message-based

A superior technique, that decouples entity communication from the entities themselves, consists in **producers** generating **messages** that are forwarded to a **message queue** and read by **consumers**. Message types can be implemented as a **tagged** `union` of lightweight structs, by using run-time polymorphism, or by using a variant type like `boost::variant`:

```
struct maximize_message { int _window_id; };
struct close_message { int _window_id; };

using message = variant<maximize_message, close_message>;
```

After modeling the union of message types, the base entity class will need to provide a `virtual` event handler method:

```
class widget
{
protected:
    virtual handle_message(const message&) { }
    // ...
}
```

Entities can create messages by enqueuing them in a shared `message_queue` object:

```
class close_button : widget
{
private:
    int _parent_id;

    void click(message_queue& mq)
    {
        mq.enqueue(message{close_message{_parent_id}})
    }

    // ...
};
```

Derived classes can override the event handler method to respond to messages:

```
class window : widget
{
private:
    int _id;

    void handle_message(const message& m) override
    {
        visit(m,
            [this](const maximize_message& x)
            {
                if(x._window_id == this->_id)
                    this->maximize();
            },
            [this](const close_message& x)
            {
                if(x._window_id == this->_id)
                    this->close();
```

```
18              });
19      }
20
21      // ...
22  };
```

This approach is versatile and can be implemented more cleverly and efficiently, although it is rarely used outside of huge-scale projects in practice. Entities never explicitly know about each other - they "subscribe" to particular message types instead and process them as they arrive. When a sender or receiver entity is destroyed, it simply stops sending or receiving messages, preventing erroneous memory accesses.

3.3.3 Advantages and disadvantages

Compared to a completely *unstructured* approach, using polymorphic objects to encode entities provides a cleaner and simpler way of managing data and logic. However, the object-oriented inheritance technique is only suitable for simple applications and games with a limited amount of entity types and a small number of active entity instances at run-time. Its main selling point is the ease of development and programming productivity.

The use of polymorphism negatively impacts performance and memory usage in comparison to a data-oriented approach, especially because it makes taking advantage of data locality and cache-friendliness impossible[2].

Using inheritance to build an entity type hierarchy lacks flexibility and, as seen in the examples, introduces significant architectural problems.

3.4 Object-oriented composition

A flexible way of solving the aforementioned *repetition* and *diamond* issues requires a point-of-view shift from a hierarchical approach to a **composition-based** one. Entities will be defined as containers of small reusable **behaviors**[3], with the following characteristics:

- Behaviors **store data** and **handle logic**;

[2] common reasons include: indirection and size overhead caused by dynamic allocation; loading unused fields in the cache. More information at [19] and [20].

[3] the term **"behavior"** is being used instead of **"component"** in order to clearly distinguish the two concepts, as the former contains and handles logic, in contrast to the latter.

- Behavior types **conform to the same interface** and polymorphically inherit from a base `behavior` class;
- Behaviors can be added and removed from entities at run-time.

From a high-level perspective, object-oriented composition looks like this:

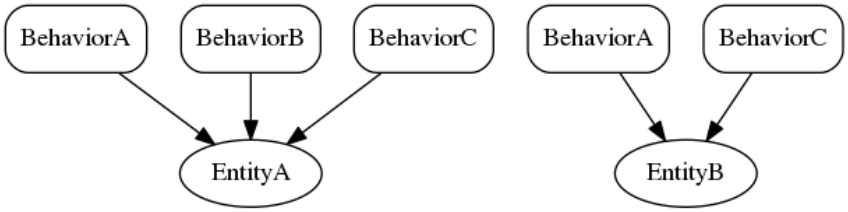


Figure 3.7: Object-oriented composition: hypotetical entity hierarchy

3.4.1 Implementation

As with the previous approach, a base class containing an application-specific `virtual` interface needs to be defined - this time the class will represent a behavior type, not an entity type.

3.4.1.1 Role-playing game

Here is an example on how the `skeleton` and `dragon` entities could be encoded using object-oriented composition:

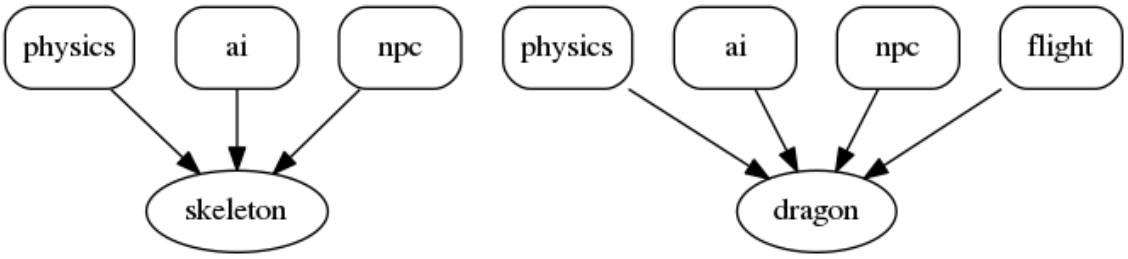


Figure 3.8: Object-oriented composition: RPG - skeleton and dragon

This approach also solves the previously encountered “diamond of death” issue:

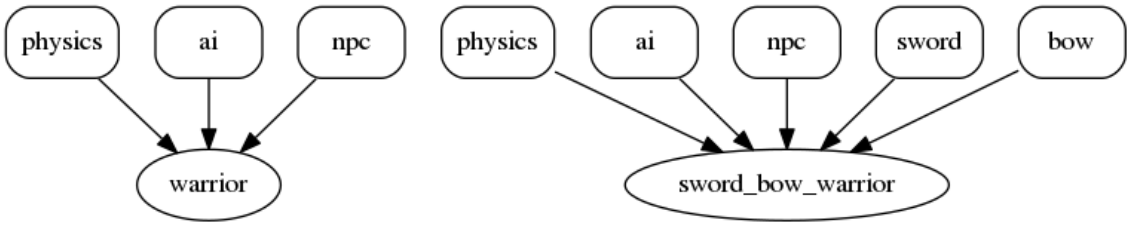


Figure 3.9: Object-oriented composition: RPG - unarmed and sword+bow warrior

The code for the base polymorphic `behavior` class closely resembles the previously seen `game_object`:

```
class behavior
{
public:
    virtual ~behavior() { }

    virtual void update(float dt) { }
    virtual void draw() { }
};


class physics : behavior { /* ... */ };
class ai : behavior { /* ... */ };
class npc : behavior { /* ... */ };
class flight : behavior { /* ... */ };
class sword : behavior { /* ... */ };
class bow : behavior { /* ... */ };
```

The implementation of `entity` provides an interface to manipulate behaviors at runtime, and stores the behaviors in a `std::vector` of `std::unique_ptr<behavior>`, in order to enable polymorphism to correctly take place:

```
class entity
{
private:
    std::vector<std::unique_ptr<behavior>> _behaviors;

public:
    template<typename T, typename... Ts>
    auto& emplace_behavior(Ts&&... xs) { /* ... */ }

    void update(float dt)
    {
        for(auto& c : _behaviors)
            c->update(dt);
    }

```

```
16      void draw()
17      {
18          for(auto& c : _behaviors)
19              c->draw();
20      }
21  };
```

Entity types **are not encoded as classes** anymore, but as **functions** that return entity instances:

```
1  auto make_skeleton(/* ... */)
2  {
3      entity result;
4      result.emplace_behavior<physics>(/* ... */);
5      result.emplace_behavior<ai>(/* ... */);
6      result.emplace_behavior<npc>(/* ... */);
7
8      return result;
9  }
```

```
1  auto make_dragon(/* ... */)
2  {
3      entity result;
4      result.emplace_behavior<physics>(/* ... */);
5      result.emplace_behavior<ai>(/* ... */);
6      result.emplace_behavior<npc>(/* ... */);
7      result.emplace_behavior<flight>(/* ... */);
8
9      return result;
10 }
```

```
1  auto make_sword_bow_warrior(/* ... */)
2  {
3      entity result;
4      result.emplace_behavior<physics>(/* ... */);
5      result.emplace_behavior<ai>(/* ... */);
6      result.emplace_behavior<npc>(/* ... */);
7      result.emplace_behavior<sword>(/* ... */);
8      result.emplace_behavior<bow>(/* ... */);
9
10     return result;
11 }
```

3.4.1.2 GUI framework

This technique solves the **repetition problem** encountered during the object-oriented inheritance-based implementation of the example GUI framework.

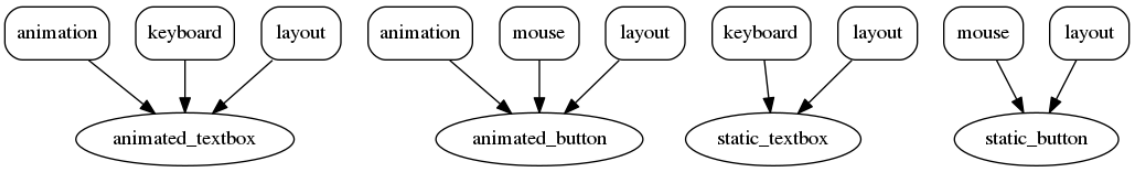


Figure 3.10: Object-oriented composition: GUI entity hierarchy

3.4.2 Communication

While entities still need to communicate with each other for the reasons described in Chapter 3, Subsection 3.3.2, this technique also may require that behaviors "talk" to one another.

Imagine implementing a `clickable` behavior for widgets:

```
1  struct clickable : behavior
2  {
3      bounding_box _bb;
4      bool _clicked;
5
6      void update(float) override
7      {
8          _clicked = mouse::overlaps(_bb);
9      }
10 };
```

Buttons and textboxes need to "ask" the `clickable` behavior whether or not they need to handle a mouse click.

3.4.2.1 Address-based

A possible approach would be either checking or asserting the existence of a behavior and then access it directly through the entity. A requirement is that behaviors store a reference to their parent entity:

```
1  class behavior
2  {
3  protected:
4      entity& _entity;
5      // ...
6  };
7
8  struct print_on_click : behavior
```

```
9  {
10     void update(float) override
11     {
12         assert(_entity.has_behavior<clickable>());
13         auto& bc = _entity.get_behavior<clickable>();
14
15         if(bc._clicked) { /* ... */ }
16     }
17 };
```

This communication method quickly becomes hard to maintain due to heavy coupling between behaviors.

3.4.2.2 Message-based

Similarly to the object-oriented approach, using lightweight messages and a mediator `message_queue` class can reduce coupling and make communication between behaviors much easier to manage. The implementation is essentially equivalent to the one previously shown: the base `behavior` type will provide a `virtual handle_message(const message&)` function that derived behavior types can override.

3.4.3 Advantages and disadvantages

Object-oriented composition is easy to implement and much more flexible than the hierarchical approach, but it's still suboptimal compared to data-oriented composition:

- No separation of data and logic is present;
- There is a significant overhead due to run-time polymorphism;
- It's still impossible to take advantage of the CPU cache.

A crucial piece of the ECS pattern, the **system**, is still missing from this technique. Its introduction will allow a clean separation of data and logic, that will lead to parallelization opportunities and possible cache-friendliness. The introduction of the system will allow to replace behaviors with logic-less **components**.

3.5 Data-oriented composition

In order to make the most of a machine's hardware, a major development paradigm shift has to be taken: enter **data-oriented design**. DOD is *all about the data*: code **has to be designed around the data** and not vice-versa. When used correctly, data-oriented design can allow applications to take advantage of parallelism and a higher percentage of cache hits[4]. Additional benefits include modularity, easier networking, and easier serialization.

To achieve all the aforementioned advantages, the following design will be used:

- Entity instances will be simple **numerical IDs**;
- **Component types** will be simple, small and logic-less;
- Component data will be **stored separately** from entities;
- Logic will be separately handled by **systems**;
  - Entities will subscribe to systems depending on their current active component types.
- A **context** *(manager)* object will **tie everything together**.
  - The various parts of the pattern will communicate with each other and with the user through the context.

An intuition for this technique is thinking about **relational database management systems** *(RDBMSs)* tables, with component types as columns, and entity instances as rows:

	ComponentA	ComponentB	ComponentC
Entity #0	X	X	
Entity #1		X	
Entity #2	X		X

The table above shows how component types can be bound and unbound from

[4] cache-friendliness is obtained by storing data contiguously in memory, loading only necessary data in the cache, and avoiding indirection. See [19] and [20] for more details.

entity instances. The previously mentioned **context** object will keep track of component availability in entity instances, thus providing an interface roughly similar to `bool context::has_component<...>(entity_id)`.

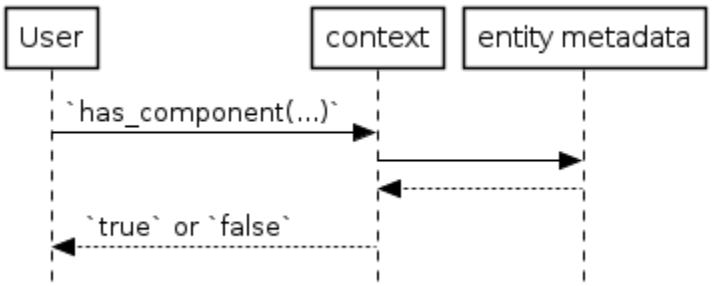


Figure 3.11: DOD: checking component type availability through context object

Knowing that an entity instance possesses a specific component type, a function that retrieves the data of the component for that particular instance will also be provided by the context object: `auto& context::get_component<...>(entity_id)`.

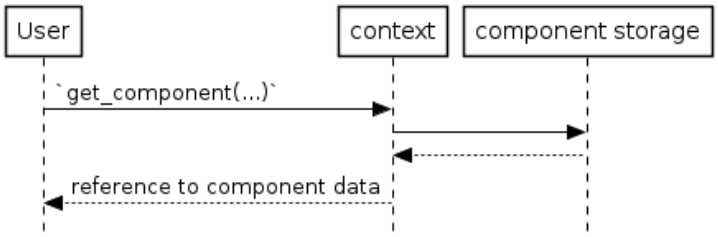


Figure 3.12: DOD: getting component instance data through context object

The role of `entity metadata` and `component storage` in the diagrams is to show the additional layer of abstraction between the user and the pattern implementation. With the introduction of the **context object** concept, it is possible to cleanly implement **systems**. They will "ask" the context to return the set of all matching entities[5], with an interface like `void context::for_entities_with<...>(f)`.

[5] in a real implementation, systems will efficiently cache the set of entities they match - the `context` will not have to continuously iterate over all existing entities.

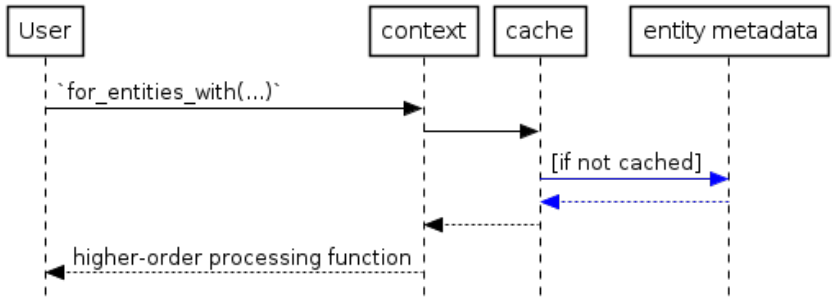


Figure 3.13: DOD: retrieving entities matching a component type set through context object

Here is a user-code example system implementation:

```
1  void example_system(context& c)
2  {
3      c.for_entities_with<a_type, c_type>([&c](auto eid)
4          {
5              auto& a_data = c.get_component<a_type>(eid);
6              auto& c_data = c.get_component<c_type>(eid);
7
8              perform_action(a_data, c_data);
9          });
10 }
```

All the logic of the application can then be defined in terms of sequential data transformations[6], which are very easy to maintain, expand and parallelize. For instance, `example_system` could be parallelized by splitting the range provided by `for_entities_with` evenly between CPU cores.

The role of the context is to decouple entities, components and systems, and provide a high level of abstraction. Component data could be stored in *arrays*, *hash tables*, or even on another machine over the network.

[6] "dataflow programming".

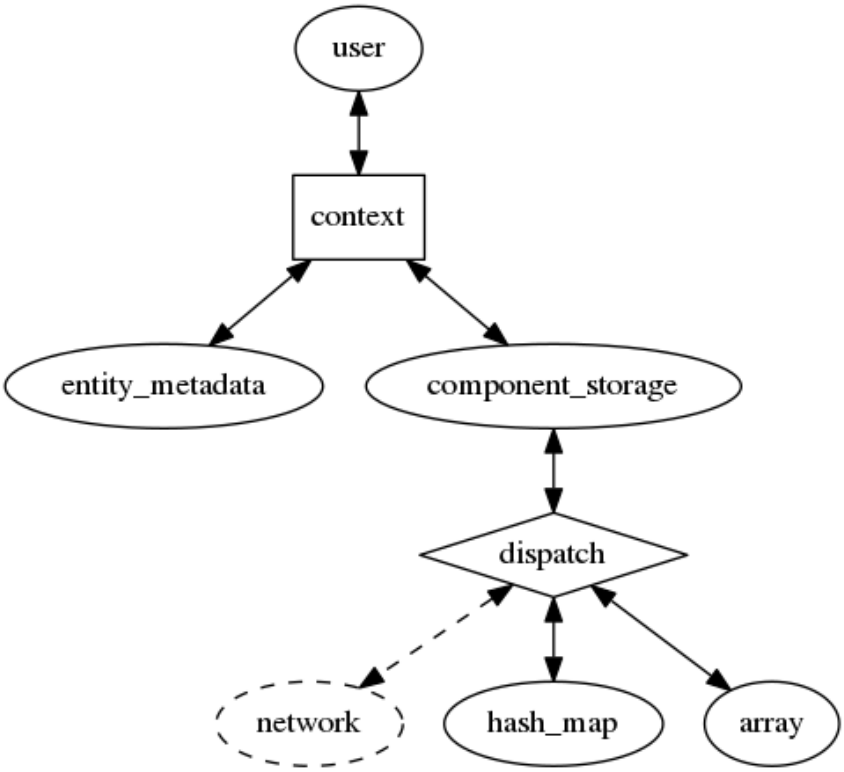


Figure 3.14: DOD: role of the context object

3.5.1 Implementation

There are some implementation details inherently related to the approach. Since entities are numerical IDs, it is sufficient to define a type alias:

```
1  using entity_id = std::size_t;
```

If all component types and system types are supposed to be known at compile-time, it is unnecessary to define base classes for them. Implementing component storage can be done in multiple ways: the most simple and straightforward way consists in using a separate array per component type:

```
1  struct component_a { /* ... */ };
2  struct component_b { /* ... */ };
3  struct component_c { /* ... */ };
4
5  constexpr auto max_entities{10000};
6
7  class component_storage
8  {
```

```
9  private:
10     std::array<component_a, max_entities> _a;
11     std::array<component_b, max_entities> _b;
12     std::array<component_c, max_entities> _c;
13
14 public:
15     template <typename TComponent>
16     TComponent& get(entity_id eid);
17 };
```

The `context` will take care of retrieving data from `component_storage`, thus making the implementation of system types extremely simple:

```
1  struct system_ac
2  {
3      void process(context& c)
4      {
5          c.for_entities_with<a_type, c_type>([&c](auto eid)
6              {
7                  /* ... */
8              });
9      }
10 };
```

The application code will roughly look like this:

```
1  int main()
2  {
3      component_storage cs;
4      context c{cs};
5
6      system_ac s_ac;
7
8      // ...create entities...
9      // ...add components...
10
11     while(running)
12     {
13         s_ac.process(c);
14     }
15 }
```

All the leftover details can be implemented in multiple ways:

- Components can be bound to entities by using a **dense bitset**, where every bit corresponds to a component type;
  - The bitsets and additional metadata can be stored in various ways - the `context` will take care of providing access to them.

- Systems can keep track of the subscribed entities by storing their IDs in an appropriate data structures;
  - The `context` can take care of matching entities to systems when a new entity is created or its components are changed.
- Components and entities can communicate with each other by allowing systems to produce outputs that can be processed by the application code.

3.5.1.1 Role-playing game

The class design is identical to the one seen in the object-oriented composition approach. Instead of using `behavior` types that handle both data and logic, they are completely split: data is stored in components, logic is handled by systems:

```
// The namespace `c` will contain all component types.
namespace c
{
    struct physics { /* ... */ };
    struct ai { /* ... */ };
    struct npc { /* ... */ };
    struct flight { /* ... */ };
    struct sword { /* ... */ };
    struct bow { /* ... */ };
}

// The namespace `s` will contain all system types.
namespace s
{
    // Processes entities with `c::physics`.
    struct physics { /* ... */ };

    // Processes entities with `c::ai` and `c::npc`.
    struct ai_npc { /* ... */ };

    // ...
}
```

Entity creation and mutation are delegated to the `context`, which takes care of bookkeeping and binding all elements together:

```
auto make_skeleton(context& c, /* ... */)
{
    auto eid = c.create_entity();
    c.emplace_component<physics>(eid, /* ... */);
    c.emplace_component<ai>(eid, /* ... */);
```

```
6       c.emplace_component<npc>(eid, /* ... */);
7
8       return eid;
9   }
```

The application code or the `context` will take care of instantiating and executing systems. Systems that do not depend on each other can be executed in parallel, and systems with no processing ordering requirements can also be internally parallelized.

3.5.1.2 GUI framework

The data-oriented implementation of the example GUI framework is conceptually identically to the RPG one. A powerful benefit of using systems to handle logic is that animation features are cleanly handled and specialized for every widget type:

```
1   namespace c
2   {
3       struct animation { /* ... */ };
4       struct layout { /* ... */ };
5       struct keyboard { /* ... */ };
6       struct mouse { /* ... */ };
7
8       // Sometimes highly-specific components can be
9       // defined to handle particular situations or to
10      // simply "tag" entities.
11      struct textbox { /* ... */ };
12      struct button { /* ... */ };
13  }
14
15  namespace s
16  {
17      // Processes entities with `c::textbox` and `c::keyboard`.
18      struct textbox { /* ... */ };
19
20      // Processes entities with `c::button` and `c::mouse`.
21      struct button { /* ... */ };
22
23      // Textbox-specific optional animations.
24      // Processes entities with `c::textbox` and `c::animation`.
25      struct anim_textbox { /* ... */ };
26
27      // Button-specific optional animations.
28      // Processes entities with `c::button` and `c::animation`.
29      struct anim_button { /* ... */ };
30  }
```

As seen from the above component and system definitions, the existence of an `animation` component in an entity will either enable or disable optional animations.

If an entity has the `animation` component, the logic will depend on the other components the entity has, making it easy to define widget-specific animation behavior.

All that is left is instantiating the `context` and the system types, then execute application-specific logic:

```
int main()
{
    context c;

    s::textbox s_textbox;
    s::button s_button;
    s::anim_textbox s_anim_textbox;
    s::anim_button s_anim_button;

    // ...create entities and components...

    while(running)
    {
        s_textbox.process(c);
        s_button.process(c);
        s_anim_textbox.process(c);
        s_anim_button.process(c);
    }
}
```

The boilerplate code shown in the code snippet above can be avoided by using advanced metaprogramming techniques. In Part 2, the way **ECST** avoids similar bookkeeping code will be analyzed.

3.5.2 Communication

3.5.2.1 Inter-component

Systems elegantly solve the problem of inter-component communication. Imagine a simple physics simulation using the following components:

```
struct position { vec3f _v };
struct velocity { vec3f _v };
struct acceleration { vec3f _v };
```

Two systems can be used to implement $v' = v + a$ and $p' = p + v$, allowing `velocity` to communicate with `acceleration` and `position` to communicate with `velocity`:

```
1  void process_acceleration(context& c)
2  {
3      c.for_entities_with<velocity, acceleration>([&c](auto eid)
4          {
5              c.get_component<velocity>(eid)._v +=
6                  c.get_component<acceleration>(eid)._v;
7          });
8  }
9
10 void process_velocity(context& c)
11 {
12     c.for_entities_with<position, velocity>([&c](auto eid)
13         {
14             c.get_component<position>(eid)._v +=
15                 c.get_component<velocity>(eid)._v;
16         });
17 };
```

3.5.2.2 Inter-system

Inter-system communication can also be very useful. Consider situations where multiple system dependency chains run in parallel: some systems may require forwarding information to the next system in the chain.

There are various ways of solving this problem:

- Systems could be stateful *(storing their outputs, so that others can access them)* and could store references to other systems. Particular care must be used in making sure that a system has finished processing before accessing its output and that its memory location does not change.

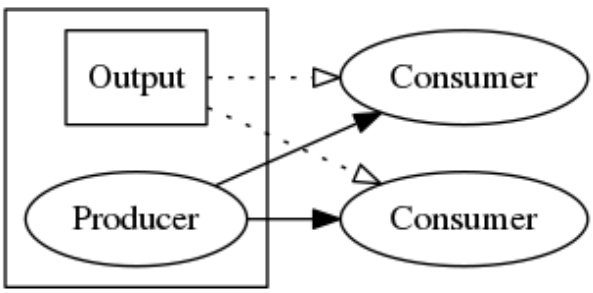


Figure 3.15: DOD communication: example stateful system communication architecture

- A **streaming** technique using queues could be used in certain situations *(depending on the implementation of the systems)* - one system would act as a

producer and other systems would act as *consumers*. The producer system would enqueue messages during entity processing - other systems would not directly process entities, instead process received messages.

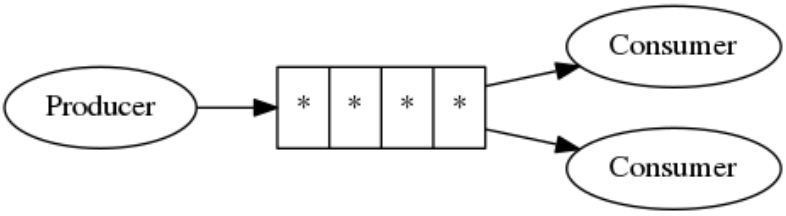


Figure 3.16: DOD communication: example streaming system communication architecture

3.5.2.3 Inter-entity

Inter-entity communication is not as useful as in the previous encoding techniques, but some particular situations may require specific entity instances to share data. *"Inter-entity"* is a misnomer, as logic-less entities cannot directly communicate together - specialized systems will have to initiate and resolve inter-entity messages. This can be achieved in multiple ways:

- Systems can produce special outputs containing the IDs of the entities that intend to communicate and a particular message. Subsequent systems will take care of processing those outputs and performing actions depending on the type/contents of the messages.
- A thread-safe queue can be accessed by systems during execution to enqueue messages. Subsequent systems can sequentially dequeue all messages and directly act upon entity instances depending on the type/contents of the messages.

Having two tightly coupled entity instances when using the Entity-Component-System pattern is a red flag: it is very likely that the coupling can be avoided by introducing new components and/or systems, or by modifying the design of the application.

3.5.3 Advantages and disadvantages

Using data-oriented composition, all the benefits of object-oriented composition are maintained, but several new advantages are achieved:

- Data and logic **are separated**;
- The code becomes **more modular**, easier to maintain and extend;
- Entities are more easily **serializable**[7] and **synchronizable** over the network;
- Entities can be processed in terms of **chained data transformations**, allowing parallelization and cache-friendliness;
- No unnecessary run-time polymorphism overhead.

One *perceived* disadvantage may be that the code is harder to reason about compared to an object-oriented approach, and inherently less abstracted. One of the objectives of this thesis is showing that, with a proper ECS library, the code is **actually easier to reason about** *(thanks to a dataflow-oriented approach)* and that high-level abstractions *(and the safety/convenience they provide)* do **not have to be sacrificed**.

In addition, data-oriented composition lends itself very nicely to multi-machine parallel computing - a real-world example is Improbable's **SpatialOS** cloud-based architecture [21], which massively parallelizes computation in a transparent way using a variant of the Entity-Component-System pattern.

3.6 Conclusion

A gradual transition from a traditional **object-oriented inheritance** entity encoding technique to a **data-oriented composition** one was shown in this chapter. The data-oriented composition approach allows an efficient and correct implementation of the **Entity-Component-System** pattern.

Every analyzed entity encoding technique has several advantages and drawbacks - the complexity of a project is the most important parameter to consider when choosing between them.

[7] the use of numerical IDs instead of pointers and the separation of data and logic make serialization trivial.

Part II

ECST

Chapter 4

Overview

ECST is a **C++14** library designed to let users implement the **Entity-Component-System** pattern in their applications leveraging **compile-time** knowledge of component types, data-structures and system dependencies to allow **automatic parallelization** of separate data transformation chains.

The source code is available on GitHub under the *Academic Free License ("AFL") v. 3.0*: https://github.com/SuperV1234/ecst.

4.1 Design

The library was designed to allow users to safely implement the ECS pattern using an **high-level of abstraction** without sacrificing **performance** and **flexibility**, using knowledge of the supported component and system types at compile-time.

4.1.1 Compile-time ECS

Every feature offered by ECST requires the instantiation of a **context** template class *(whose role is the management of the entire pattern)* which provides the user with a high-level interface to interact with entities, components and systems. The context object requires **prior knowledge** of all **component types** and **system types** at compile-time. Additionally, options regarding multithreading support, system scheduling and entity limits can be specified before the instantiation of a context.

Due to these factors, ECST is **not a data-driven** Entity-Component-System library. If the target application domain requires dynamic run-time composition and flexibility, it is sensible *(and recommended)* to use ECST with an additional data-driven ECS library[1].

- All component and system types must be known at compile-time;
- Entities can be created, destroyed, tracked and mutated at run-time.

Some disadvantages of this approach include:

- Longer and potentially-unreadable errors;
- Increased compilation times.

The implementation details regarding the definition and usage of compile-time settings will be explored in Chapter 6 and Chapter 7. To make the following points simpler to understand and to give the readers an idea of the previously mentioned limitations and features, the code snippet below will illustrate how context objects are configured and instantiated in the user code.

4.1.1.1 Code example: settings definition

Imagine a 2D particle simulation composed by:

- Three component types: **Position**, **Velocity**, and **Acceleration**.
- Two system types: **Velocity** and **Acceleration**.

Component types are defined as simple **POD** *(plain-old-data)* `struct` types.

```
1  // The namespace `c` will contain all component types.
2  namespace c
3  {
4      struct position { /* ... */ };
5      struct velocity { /* ... */ };
6      struct acceleration { /* ... */ };
```

[1] ECST can be used to implement engine-level components and systems, while an additional data-driven library provides run-time flexibility and extensibility. Integration with ECST could be achieved by creating a "bridge" component that links entities between the two libraries.

```
}
```

System types are declared as `struct` types that will eventually be defined alongside their processing logic.

```
//The namespace `s` will contain all system types.
namespace s
{
    struct velocity;
    struct acceleration;
}
```

ECST makes heavy use of the **type-value-encoding** *(a.k.a. dependent typing)* meta-programming paradigm. It is necessary to define wrapper `constexpr` **tag** values that will store the type information of components and systems: tags are be used to greatly simplify both the user interface *(as accessing template methods in C++ requires the cumbersome* `.template` *disambiguation syntax)* and implementation code.

Component tags are defined using `ecst::tag::component::v` :

```
// The namespace `ct` will contain all component tags.
namespace ct
{
    constexpr auto position =
        ecst::tag::component::v<c::position>;

    constexpr auto velocity =
        ecst::tag::component::v<c::velocity>;

    constexpr auto acceleration =
        ecst::tag::component::v<c::acceleration>;
}
```

System tags are defined using `ecst::tag::system::v` :

```
// The namespace `st` will contain all system tags.
namespace st
{
    constexpr auto velocity =
        ecst::tag::system::v<c::velocity>;

    constexpr auto acceleration =
        ecst::tag::system::v<c::acceleration>;
}
```

The verbosity of the code shown above can be avoided through the use of convenient

preprocessor macros. The definition of tag objects only needs to occur once in the entire user codebase.

Having defined all required types and tags, the next step will consist in defining *signatures*:

- **Component signatures** bind component tags to **storage policies**.
- **System signatures** bind system tags to multiple settings which will be analyzed in depth in Chapter 7, Subsection 7.1.2: **parallelization policies** and **dependencies** can be found among them.

Signatures are stored in compile-time type lists called **signature lists**, which must be forwarded to the context creation, in order to instantiate a context object.

The `make_csl()` function will create a component signature for every previously-defined component tag, and will return a component signature list holding all of them.

```
1  constexpr auto make_csl()
2  {
3      // Component signature namespace aliases.
4      namespace sc = ecst::signature::component;
5      namespace slc = ecst::signature_list::component;
6
7      // Store `c::acceleration`, `c::velocity` and `c::position` in
8      // three separate contiguous buffers (SoA).
9      constexpr auto cs_acceleration =
10         sc::make(ct::acceleration).contiguous_buffer();
11
12     constexpr auto cs_velocity =
13         sc::make(ct::velocity).contiguous_buffer();
14
15     constexpr auto cs_position =
16         sc::make(ct::position).contiguous_buffer();
17
18     // Build and return the "component signature list".
19     return slc::make(cs_acceleration, cs_velocity, cs_position);
20 }
```

The `make_ssl()` function will create a system signature for every previously-defined system tag, and will return a system signature list holding all of them.

```
1  constexpr auto make_ssl()
2  {
3      // System signature namespace aliases.
```

```
    namespace ss = ecst::signature::system;
    namespace sls = ecst::signature_list::system;

    // Acceleration system signature.
    constexpr auto ss_acceleration =
        ss::make(st::acceleration)
            .parallelism(split_evenly_per_core)
            .read(ct::acceleration)
            .write(ct::velocity);

    // Velocity system signature.
    constexpr auto ss_velocity =
        ss::make(st::velocity)
            .dependencies(st::acceleration)
            .parallelism(split_evenly_per_core)
            .read(ct::velocity)
            .write(ct::position);

    // Build and return the "system signature list".
    return sls::make(ss_acceleration, ss_velocity);
}
```

The final setup step consists in the definition of a `constexpr` **context settings** instance, which will be used to instantiate an ECST context.

```
constexpr auto context_settings =
    ecst::settings::make()
        .component_signatures(make_csl())
        .system_signatures(make_ssl())

auto context =
    ecst::context::make(context_settings);
```

Note that this example skipped many possible configuration options and implementation and design details for the current settings definition approach: the goal of the code snippets above is to clarify that component and system types *need* to be known at compile-time and *how* that information is passed to an ECST context. The knowledge provided by `context_settings` will be used to **generate** the following elements at **compile-time**:

- Storage for **entity metadata**, **component data**, and **systems**;
- An implicit **system dependency graph**, which is used to automatically run different systems in parallel.

Easily interchangeable compile-time options also allow developers to experiment with different data layouts and scheduling policies without having to modify the application code.

4.1.2 Customizability

The compile-time-driven nature of the library lends itself to **policy-based design** for user-configurable options. In line with the *"pay for what you use"* C++ philosophy, the specified policies are taken into account in various ways to optimize run-time performance and memory usage.

All featured options will be analyzed in Chapter 7. As a simple example, the following code snippet that generates two different context instances can help understand the power of policy-based settings:

4.1.2.1 Code example: policy-based customization

```
constexpr auto context_settings_0 =
    ecst::settings::make()
        .allow_inner_parallelism()
        .fixed_entity_limit(ecst::sz_v<10000>);

constexpr auto context_settings_1 =
    ecst::settings::make()
        .disallow_inner_parallelism()
        .dynamic_entity_limit()
        .scheduler(cs::scheduler<user_defined_scheduler>);
```

Note how easy and intuitive it is to change major library components and affect the flexibility and performance of user applications. This pattern is not only used in context settings definition, but also in component and system signatures *(e.g. inner parallelism policies or system dependencies)*.

The approach also favours **test-driven-development** and application code **transparency**: tests and benchmarks covering various combinations of policies *(either exhaustively or with a fuzzy approach)* can be easily generated by using nested compile-time loops over desired policy lists, thus providing information regarding the correctness and performance of the user code.

4.1.3 Abstraction, user-friendliness, and safety

A common misconception regarding **data-oriented design** is that it's necessary to sacrifice **encapsulation** and to **reduce the level of abstraction** in order to achieve performance and cache-friendliness. The following beliefs are commonly found in online discussions:

- High-level multi-paradigm languages like **C++14** or **D** are not suitable for high-performance DOD because they inherently drive programmers to design highly-abstracted code that is not closely mapped to the machine hardware. The origin of this belief can be attributed to the abuse of OOP techniques, such as run-time polymorphism and encapsulation, which tend to increase the level of abstraction, with the side-effect of harming data locality and leading to suboptimal memory layouts;
- Every abstraction has an intrinsic cost *("there is no such thing as cost-free abstractions").*

These misbeliefs may result in the fallacy that performant DOD code must be devoid of high-level abstractions.

ECST aims to *counter this fallacy* by providing a high-level interface that does not sacrifice run-time performance thanks to C++14 **cost-free abstractions**. Benefits of a highly-abstracted data-oriented ECS library include:

- **User-friendly** and **transparent** syntax. No explicit storage or pointer management is required, and application code is independent of user-defined settings and policies;
- Higher **safety**, as many development mistakes regarding dependencies and data access can be caught at compile-time;
- Easier **testability**, due to the fact that abstracting settings and policies allows tests to run comprehensively over a wide set of option combinations;
  - Closely related, the possibility of quickly **experimenting** with different strategies is also a major benefit of a higher abstraction level.
- **Encapsulation**, **reusability**, and fulfillment of the **DRY** principle: abstracting and templatizing storage strategies and policies allows code to be conveniently reused.

4.1.3.1 Syntax-level transparency

One important design goal of ECST is allowing the user to experiment with different policies, schedulers and execution methods **without having to explicitly change**

the application code. This objective is achieved through the use of **proxy** objects and heavily-templatized type-value-encoding implementation code. The result is a **syntax-level transparency** that allows the application code to be completely independent of compile-time policies/strategies.

Proxies and transparency implementation details will be covered in Chapter 11 - a simple code example will illustrate one possible use of syntax-level transparency through proxies.

4.1.3.2 Code example: transparency through proxies

Here's a possible implementation of the `s::velocity` system declared in Subsection 4.1.1:

```
1  struct velocity
2  {
3      template<typename TData>
4      void process(TData& data)
5      {
6          data.for_entities([](auto eid)
7          {
8              auto& position =
9                  data.get(ct::position, eid)._v;
10
11             const auto& velocity =
12                 data.get(ct::velocity, eid)._v;
13
14             position += velocity;
15         });
16     }
17 };
```

The `velocity` system `struct` contains a `process` template method that takes a single `TData& data` *lvalue reference* as its argument *(note that `process` is not a special method that is specifically recognized by ECST)*. `data` is a **data proxy** object, which abstracts the operations of a system behind a **safe and transparent** interface.

By invoking the `data.for_entities` method with a callable object, it is possible to iterate over the entities subscribed to the system, and perform some actions on the component they own. In this case, we're accessing the `c::position` component through a mutable reference, and the `c::velocity` component through a `const` reference, in order to move the particle by its velocity.

The interesting thing is that this syntax is **completely independent** of the system

settings and execution policies - the system's dependencies or parallelism policies do not matter to the data proxy. Other things to take notice of are:

- The storage policy of the components is irrelevant. `c::position` and `c::velocity` could be stored in the same contiguous buffer, or one of them could be in an hash-map while the other one is in a machine on the other side of the world *(storing a component in a different continent could slightly affect the performance of your application)*;
- Getting a mutable or `const` reference to a component is **statically checked** at compile-time, producing an error in case the component access does not fulfill what was specified in the system signature;
- `process` is not special, and not limited to a single argument - additional data could be passed to the method and captured in the lambda. This will be covered in detail in Chapter 8 - here's a possible way of invoking `s::velocity::process`:

```
1  namespace sea = ecst::system_execution_adapter;
2
3  context.step([](auto& proxy)
4  {
5      proxy.execute_systems(
6          sea::t(st::velocity).for_subtasks(
7              [](auto& s, auto& data)
8              {
9                  s.process(data);
10             })
11     );
12 });
```

4.1.4 Multithreading model

The library was designed with user-friendly multithreading as one of the primary goals, which is achieved by providing two levels of parallelism that considerably increase application performance in the presence of multiple CPU cores: **"outer parallelism"** and **"inner parallelism"**.

Multithreading support is enabled by default, but can be switched off completely or only in particular systems.

4.1.4.1 Outer parallelism

"Outer parallelism" is the term used in ECST which defines the concept of running multiple systems that do not depend on each other in parallel. Its implementation details will be analyzed in Chapter 10. Conceptually, an **implicit directed acyclic graph** is created at compile-time thanks to the knowledge of system dependencies. The execution of the implicit DAG is handled by a **system scheduler** type specified during settings definition.

Consider the previously defined system signatures - an implicit DAG isomorphic to the one below will be generated by ECST *(arrows between nodes should be read as "depends on")*:

Figure 4.1: ECST multithreading: example outer parallelism DAG #0

The graph makes it obvious that no outer parallelism can take place here. Increasing the number of component and system types introduces separate dependency chains - imagine the addition of **Growth**, **Shape**, **Rotation** and **Rendering** systems for a graphical particle simulation:

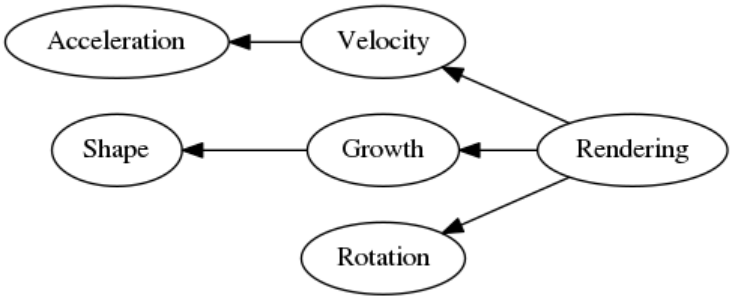


Figure 4.2: ECST multithreading: example outer parallelism DAG #1

The DAG now shows three separate dependency chains:

- Rendering → Velocity → Acceleration;
- Rendering → Growth → Shape;
- Rendering → Rotation.

The chains starting with **Acceleration**, **Shape** and **Rotation** can be executed in parallel. The **Rendering** system will wait until all three chains have been successfully executed, then it will process its subscribed entities.

Note that the user does not have to specify these chains anywhere - they are implicitly created thanks to the dependencies described during system signature definition.

4.1.4.2 Inner parallelism

Other that running separate systems in parallel, ECST supports splitting a single system into multiple **sub-tasks**, which can be executed on separate threads. Many systems, such as the ones that represent functionally pure computations, do not contain *side-effects* that modify their own state or that define interactions between the subscribed entities: these are prime examples of **"embarrassingly parallel"** computations. In contrast, some systems *(e.g. broad-phase collision spatial partitioning)* require processing their subscribed entities on a single thread, in order to update data structures without explicit locking mechanisms[2].

The aforementioned **Velocity** and **Acceleration** systems are suited for inner parallelism. The user can enable inner parallelism when defining system signatures, also choosing an **inner parallelism strategy** that can be generated by composing multiple strategies together *(e.g. "split into 4 sub-tasks only if subscribed entity count is more than 100000").*

Imagine applying a *"split into 4 sub-tasks"* strategy to **Velocity** and **Acceleration** - the resulting DAG would look as follows:

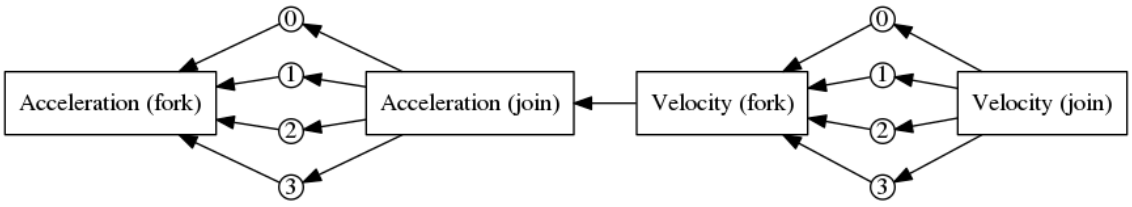


Figure 4.3: ECST multithreading: example inner parallelism DAG

This allows the 0..4 sub-tasks to run in parallel. Commonly, **inner parallelism executors**, which implement inner parallelism policies, simply split the subscribed

[2]some systems need to mutate a data structure during execution. Splitting them in separate subtasks requires *explicit locking* or a *thread-safe* data structure to avoid race conditions. It is often more efficient to simply run these systems in a single subtask.

entities of a system evenly between the generated subtasks. The *"fork"* and *"join"* nodes present in the DAG are implicit - they can however be handled by users thanks to **system execution adapters** *(described in the "advanced features" chapter)*, allowing to code execution before and/or after the execution of a system's sub-tasks.

Chapter 5

Architecture

Before analyzing the code and techniques used to implement ECST, the architecture of the library will be examined.

A very high-level view of ECST's architecture can be illustrated as such:

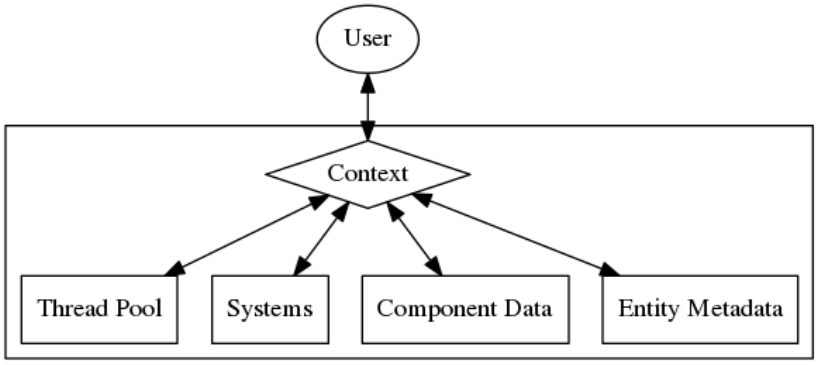


Figure 5.1: ECST architecture: high-level overview

By looking at the diagram, it is obvious that the roles of the context are:

- Providing a **friendly interface** between the user and the library;
- **Decoupling** entities, components, and systems.

5.1 Context

In the current version of the library, the `context` stores entity, system and component data.

- The composition of the **entity metadata storage** and of the **component data storage** is called `main_storage`.
- The element managing systems, threads, and scheduling is called the `system_manager`.

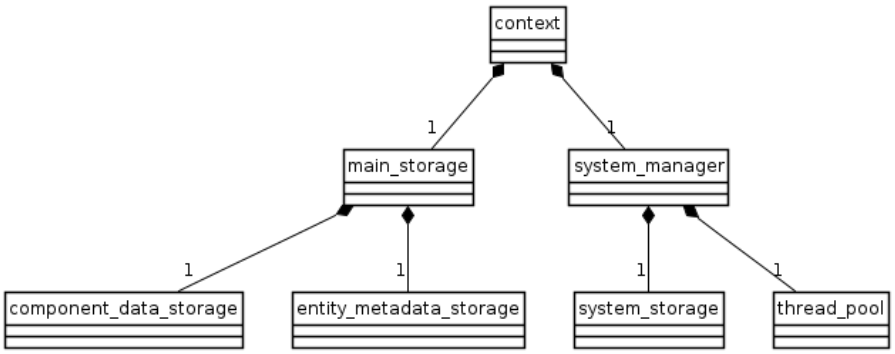


Figure 5.2: ECST architecture: context

5.1.1 Entity metadata storage

The **entity metadata storage** contains `metadata` instances for every entity, which are used to:

- Keep track of the component types an entity possesses;
- Map entities to handles and check their validity.

It also manages unused and used entity IDs.

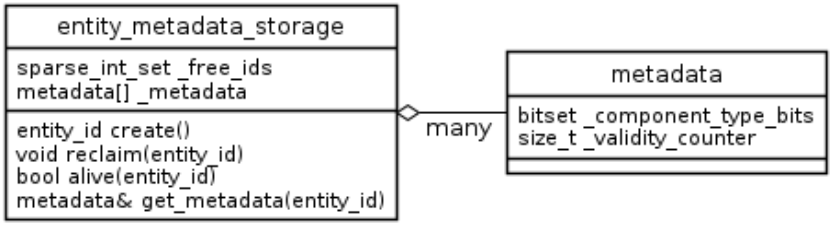


Figure 5.3: ECST architecture: entity metadata storage

5.1.2 Component data storage

The **component data storage** contains `chunk` instances for every component signature type. **Chunks** store the data of one or more component types with a user-specified strategy *(e.g. contiguous buffer or hash map)* and provide a way to retrieve references to component data given entity IDs.

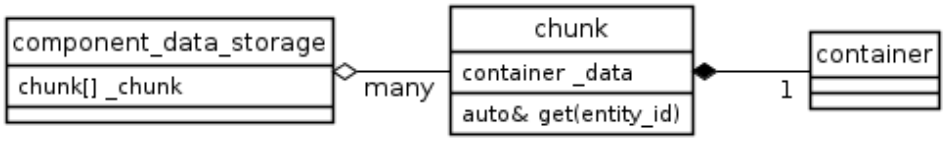


Figure 5.4: ECST architecture: component data storage

5.1.3 System manager

The system manager contains:

- A `thread_pool` instance, used to efficiently execute system logic in separate threads;
- A `system_storage`, which stores an `instance` for every system signature type.
  - **Instances** store an instance of the user-provided system type and system metadata.

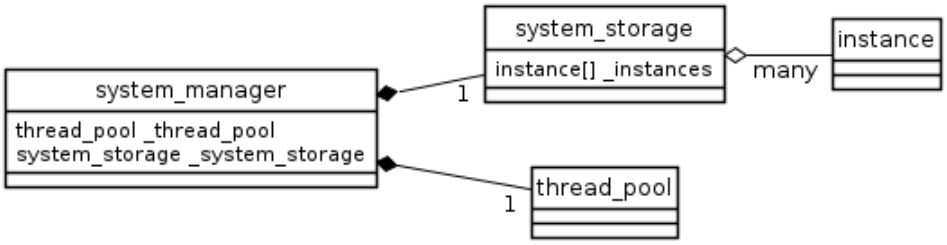


Figure 5.5: ECST architecture: system manager

5.1.3.1 Instances

Instances wrap user-provided systems, storing an instance of the "real" system type and additional metadata:

- A `state_manager`, containing `state` instances;
  - Every subtask executed by an `instance` has its own `state`, which contains eventual *system output*, *deferred functions*[1], and IDs of entities to reclaim.
- A *sparse integer set*[2] tracking the entity IDs subscribed to the instance;
- A *dense bitset*[3] representing the set of component types required for system subscription.

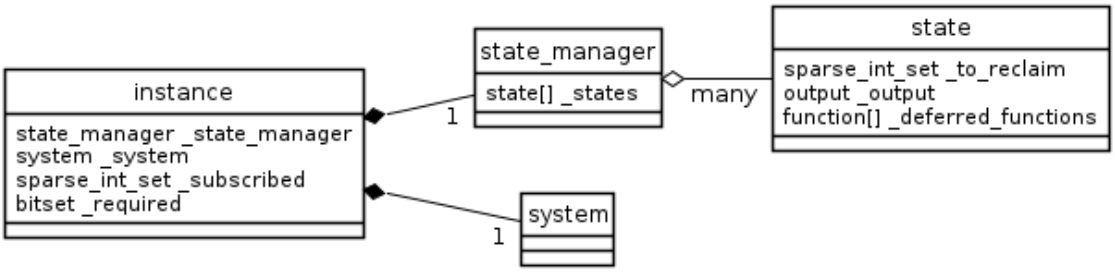


Figure 5.6: ECST architecture: system instance

[1] used to delay the execution of functions in a later synchronous step.

[2] "sparse integer sets" are very efficient data structures for the management of entity IDs. They are covered in Chapter 14, Section 14.1.

[3] `std::bitset`.

Chapter 6

Metaprogramming

Before diving into Entity-Component-System related implementation details, the **metaprogramming** techniques used throughout the library will be covered in this chapter, as understanding them is a prerequisite for the analysis of ECST's modules.

6.1 Boost.Hana

Boost.Hana [22] is an astonishing modern *header-only* C++14 metaprogramming library created by Louis Dionne that uses the **type-value encoding** paradigm *(a.k.a. dependent typing [23])* - it is heavily used throughout ECST's implementation. By wrapping types in values, Hana allows users to perform both type-level computations and heterogeneous computations using natural syntax[1] and with minimal compilation time overhead.

A simple example, taken from the original documentation, shows the idea behind type-value encoding:

```
1  auto animal_types = hana::make_tuple(
2      hana::type_c<Fish*>, hana::type_c<Cat&>, hana::type_c<Dog>);
3
4  auto no_pointers = hana::remove_if(animal_types, [](auto a) {
5      return hana::traits::is_pointer(a);
6  });
7
8  static_assert(no_pointers ==
9      hana::make_tuple(hana::type_c<Cat&>, hana::type_c<Dog>), "");
```

[1] syntax that "looks like" regular run-time computation syntax.

As seen in the code snippet above, types can be manipulated as values *(even using lambdas)*. Hana provides a huge number of powerful algorithms and utilities that work both on types and "traditional" values.

As an example, component and system signature lists are implemented as `boost::hana::basic_tuple` instances containing the user-specified signature types wrapped in `boost::hana::type_c` instances, which can be passed around like any other value and easily manipulated.

6.2 Tags

6.2.1 Motivation and usage

A very large number of methods in the library is parametrized on component types and system types. User code calling those methods would normally require the constant and inelegant use of the `instance.template method<...>(...)` template disambiguation syntax[2]. **Tags** are designed to solve this problem.

Examples will be used in order to easily explain the role of tags. Here is an hypotetical set of component and system types:

```
1  namespace c
2  {
3      struct position { /* ... */ };
4      struct velocity { /* ... */ };
5      struct acceleration { /* ... */ };
6  }
7
8  namespace s
9  {
10     struct velocity { /* ... */ };
11     struct acceleration { /* ... */ };
12 }
```

Imagine a function that creates a particle using the components listed above using "traditional" template method calling syntax:

[2]such syntax negatively impacts the readability of the code and is essentially avoidable boilerplate. It is required due to ambiguous parsing for the `<` token, which could be interpreted both as an *"angle bracket"* or as a *less-than operator*. Details can be found on cppreference.

```
auto make_particle = [](auto& proxy)
{
    auto eid = proxy.create_entity();

    proxy.template add_component<c::position>(eid);
    proxy.template add_component<c::velocity>(eid);
    proxy.template add_component<c::acceleration>(eid);

    return eid;
};
```

In order to prevent the mandatory `.template` disambiguation syntax and to treat component and system types as values, ECST provides **component tags** and **system tags**. **Tags** are `constexpr` wrappers that store the type information of components and systems in values, allowing them to be conveniently passed to implementation and interface functions with a natural syntax.

Tags need to be defined by the user, with the following syntax[3].

```
// The namespace `ct` will contain all component tags.
namespace ct
{
    constexpr auto position =
        ecst::tag::component::v<c::position>;

    constexpr auto velocity =
        ecst::tag::component::v<c::velocity>;

    constexpr auto acceleration =
        ecst::tag::component::v<c::acceleration>;
}
```

It is a good practice to separate *component types*, *system types*, *component tags* and *system tags* in separate namespaces.

```
// The namespace `st` will contain all system tags.
namespace st
{
    constexpr auto velocity =
        ecst::tag::system::v<c::velocity>;

    constexpr auto acceleration =
        ecst::tag::system::v<c::acceleration>;
}
```

Afterwards, calling template methods becomes much more natural:

[3] preprocessor macros can be used to reduce required boilerplate code.

```
auto make_particle = [](auto& proxy)
{
    auto eid = proxy.create_entity();

    proxy.add_component(ct::position, eid);
    proxy.add_component(ct::velocity, eid);
    proxy.add_component(ct::acceleration, eid);

    return eid;
};
```

Manipulating and storing tags is also easier, both in user and implementation code, resulting in a more maintainable and extensible codebase.

6.2.2 Implementation

All tag-related types and functions are declared in the `ecst::tag` namespace. Both component tags and system tags are implemented in the same way: a `struct` deriving from `boost::hana::type` is defined, that can be immediately used in any Boost.Hana algorithm.

```
namespace impl
{
    template <typename T>
    struct tag_impl : public boost::hana::type<T> { };
}
```

A `constexpr` variable called `v` *(standing for "value")* is provided as a convenient way for the user to define tags:

```
template <typename T>
constexpr auto v = impl::tag_impl<T>{};
```

The types encoded inside tags can be accessed using `ecst::mp::unwrap` [4], which is a type alias for types stored inside `boost::hana::type`:

```
namespace mp
{
    template <typename T>
    using unwrap = typename T::type;
}
```

[4] the `ecst::mp` namespace contains metaprogramming-related utilities.

Converting a traditional template method to a tag-accepting one is a straightforward process: the original method is hidden using the `private` access specifier - the new one will call it by *unwrapping* the tag:

```
struct example
{
private:
    template <typename TComponent>
    auto& access_component(entity_id);

public:
    template <typename TComponentTag>
    auto& access_component(TComponentTag, entity_id eid)
    {
        return access_component<mp::unwrap<TComponentTag>>(eid);
    }
};
```

Checking if a type or a value is a tag is also possible thanks to the following utilities:

```
namespace impl
{
    template <typename T>
    constexpr auto is_tag_impl =
        mp::is_specialization_of_v<tag_impl, T>;

    struct valid_t
    {
        template <typename... Ts>
        constexpr auto operator()(Ts...) const
        {
            return mp::list::all_variadic(is_tag_impl<Ts>...);
        }
    };
}

// Evaluates to true if all `xs...` are component tags.
constexpr impl::valid_t valid{};
```

Note that `valid` is implemented as a `constexpr` value *(instance of `impl::valid_t`)* and not as a regular function. This pattern is used throughout ECST, due to the fact that functions implemented as `constexpr` instances of callable objects can be easily used as arguments to other functions without the need of a lambda wrapper. This is the case for the `tag::component::is_list` and `tag::system::is_list` functions, that return whether or not the passed argument is a list of tags:

```
template <typename T>
constexpr auto is_list(T x)
{
    return boost::hana::all_of(x, valid);
}
```

6.3 Option maps

6.3.1 Motivation and usage

User-provided compile-time settings are a vital part of ECST: in order to allow users to set options in a convenient and clear way, **option maps** were implemented using `boost::hana::map` instances and method chaining. Here is an example of user code that creates a *system signature* for `s::acceleration`:

```
constexpr auto ss_acceleration =
    ss::make(st::acceleration)
        .parallelism(split_evenly_per_core)
        .read(ct::acceleration)
        .write(ct::velocity);
```

The code snippet above defines `ss_acceleration` to be a system signature for `s::acceleration` with the following settings, known at **compile-time**:

- Use the `split_evenly_per_core` *inner parallelism strategy*;
- Use the `c::acceleration` component (read-only);
- Use the `c::velocity` component (mutable).

The options are provided by the user by chaining methods such as `.parallelism(...)` and `.read(...)`. The calls can be freely re-ordered, and if the same method is accidentally called twice, a compile-time error will be generated.

Defining settings using the pattern above is possible thanks to the `ecst::mp::option_map` compile-time data structure.

6.3.2 Implementation

Conceptually, an option map is a compile-time associative container with the following properties:

- Keys are values fulfilling the Boost.Hana `Comparable` and `Hashable` concept;
  - A set of keys is composed by `constexpr` `boost::hana::integral_constant` instances, with incrementing values.
- The keys are associated with compile-time pairs of an user-defined type and `boost::hana::bool_c`.
  - The user-defined type is option-specific;
  - The boolean integral constant is used to mark the option as *"already set"*, in order to prevent accidental multiple method calls.

Option maps are implemented as *immutable data structures*: performing operations on them returns a new option map with the desired changes.

```
1  template <typename TMap>
2  class option_map
3  {
4  private:
5      TMap _map;
6
7  public:
8      template <typename TKey>
9      constexpr auto at(const TKey& key) const;
10
11     template <typename TKey, typename T>
12     constexpr auto add(const TKey& key, T&& x) const;
13
14     template <typename TKey, typename T>
15     constexpr auto set(const TKey& key, T&& x) const;
16 };
```

The `_map` field is a `boost::hana::map` instance. The `set` method returns a new `option_map` instance, adding `key -> (x, boost::hana::false_c)` to `_map`. The `add` method returns a new `option_map` instance, setting the value at `key` to `(x, boost::hana::true_c)` - it statically asserts the bool at `key` to be false, to prevent users from accidentally modifying a previously set option.

Using an option map to implement compile-time settings with method chaining requires the following elements:

- A class that contains an `option_map` instance and provides a clean method chaining interface;

- A set of keys representing the available options;
- An initial set of default options.

As an example, the implementation of system signatures will be analyzed.

6.3.2.1 Example: system signature settings

The class that contains the option map, the interface, and the system tag is declared as such:

```
template <typename TTag, typename TOptionMap>
class system_signature;
```

It is default-instantiated using the `ecst::signature::system::make(...)` function:

```
template <typename TSystemTag>
constexpr auto make(TSystemTag)
{
    constexpr auto default_options =
        mp::option_map::make()
            .add(keys::parallelism, ips::none::v())
            .add(keys::dependencies, hana::make_tuple())
            .add(keys::read_components, hana::make_tuple())
            .add(keys::write_components, hana::make_tuple())
            .add(keys::output, no_output);

    return system_signature<TSystemTag,
        decay_t<decltype(default_options)>>{};
}
```

The keys shown above are `constexpr` instances of Hana integral constants with unique values:

```
namespace keys
{
    constexpr auto parallelism = hana::int_c<0>;
    constexpr auto dependencies = hana::int_c<1>;
    constexpr auto read_components = hana::int_c<2>;
    constexpr auto write_components = hana::int_c<3>;
    constexpr auto output = hana::int_c<4>;
}
```

After instantiating a default `system_signature` by calling `signature::system::make`, the user can modify options by using its methods, which return updated copies of

the signature. After calling any number of unique methods, the final expression will evaluate to an instance of `system_signature` containing all user-desired settings.

```
template <typename TTag, typename TOptionMap>
class system_signature
{
private:
    TOptionMap _option_map;

    template <typename TKey, typename T>
    constexpr auto change_self(const TKey& key, T x) const
    {
        auto new_map = _option_map.set(key, x);
        return system_signature<TTag, decay_t<decltype(new_map)>>{};
    }

public:
    template <typename TParallelismOption>
    constexpr auto parallelism(TParallelismOption x) const
    {
        return change_self(keys::parallelism, x);
    }

    // ...
};
```

Upon passing the final customized system signature to the context, ECST can retrieve the options provided by the user by querying `_option_map`. This pattern is currently used to implement **system signatures**, **component signatures** and **context settings**.

6.4 Other techniques and algorithms

A number of other metaprogramming techniques and compile-time algorithms are used throughout ECST:

- A **breadth-first traversal** algorithm is used to find and count dependencies in isolated system chains - it is covered in Chapter 14, Section 14.4;
- Compile-time **tuple element iteration** is used to build *component type bitsets*, to start parallel system execution and to run tests with various combinations of settings. See Chapter 14, Section 14.2 for more details;

- **SFINAE**, `std::enable_if_t`, and generic lambdas with trailing return types are used together with `boost::hana::overload` to implement *system execution adapters*, *refresh event handling*, and *generic system execution*. These techniques will be covered in Chapter 8 and Chapter 12.

Chapter 7

Compile-time settings

After including ECST, the user must define some mandatory compile-time settings, required to instantiate a `context` :

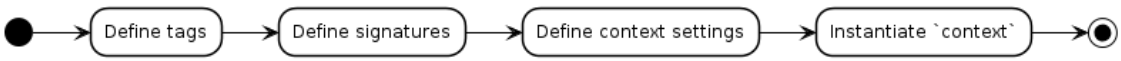


Figure 7.1: ECST compile-time settings: mandatory options

Tag definition was previously described in Chapter 6, Section 6.2. **Signatures** and **context settings** will be covered in the following sections.

7.1 Signatures

Signatures are `constexpr` values containing compile-time options. They are implemented using option maps, covered in Chapter 6, Section 6.3. There are two kinds of signatures: **component signatures** and **system signatures**.

7.1.1 Component signatures

Component signatures are used to *bind* storage strategies with component types. Multiple component tags can be bound to a specific storage strategy. Users can implement their own storage strategies *(briefly explained in Chapter 9, Subsection 9.1.1)*. The `contiguous_buffer` strategy is available by default and allows users to store components in contiguous memory locations.

7.1.1.1 SoA

Creating a `contiguous_buffer` signature per component type results in a **SoA** *(structure of arrays)* storage layout:

```
namespace cs = ecst::signature::component;

constexpr auto cs_acceleration =
    cs::make(ct::acceleration).contiguous_buffer();

constexpr auto cs_velocity =
    cs::make(ct::velocity).contiguous_buffer();

constexpr auto cs_position =
    cs::make(ct::position).contiguous_buffer();
```

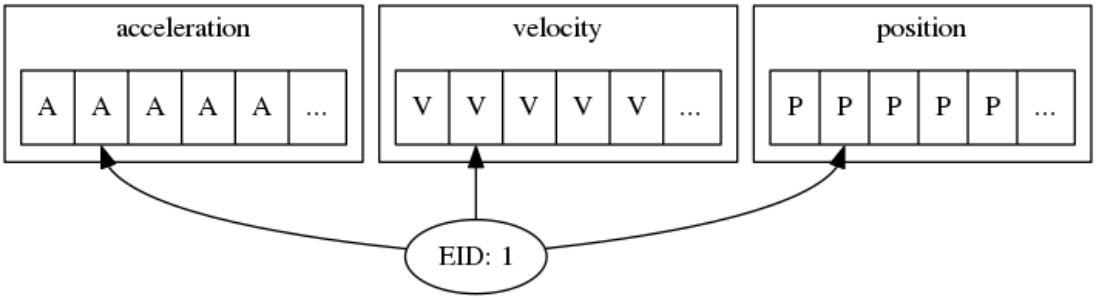


Figure 7.2: ECST compile-time settings: high-level view of SoA component storage layout

7.1.1.2 AoS

Creating a `contiguous_buffer` signature with multiple component types results in a **AoS** *(array of structures)* storage layout:

```
constexpr auto cs_physics = signature::component::make(
    ct::acceleration, ct::velocity, ct::position)
        .contiguous_buffer();
```

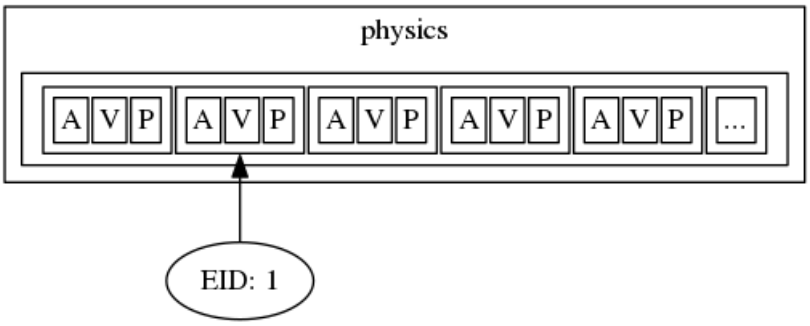


Figure 7.3: ECST compile-time settings: high-level view of AoS component storage layout

7.1.2 System signatures

System signatures are used to define the following system settings:

- **Inner parallelism policy**: either allows the system to run in multiple threads *(subtasks)* using a specific strategy or forces the system to run in a single thread;
- **List of dependencies**: list of systems whose execution needs to be completed before allowing the current one to run. Used to build the previously mentioned outer parallelism DAG;
- **Accessed component types**: components types mutated or read by the system. Used to subscribe matching entities to the system and to allow and verify component access in system implementation;
- **Output type**: type of data produced by every subtask, if any.

An example of a system signature definition is given below:

```
1 constexpr auto ss_collision =
2     signature::system::make(st::collision)
3         .parallelism(split_evenly_per_core)
4         .dependencies(st::spatial_partition)
5         .read(ct::circle)
6         .write(ct::position, ct::velocity)
7         .output(ss::output<std::vector<contact>>);
```

7.1.3 Signature lists

Signature lists are compile-time lists of system signatures, that are used to group all defined component signatures and system signatures together, in order to pass them to the context settings definition.

They are implemented using `boost::hana::basic_tuple`. The user creates them using the following syntax:

```
constexpr auto make_csl()
{
    constexpr auto cs_acceleration = /* ... */;
    constexpr auto cs_velocity = /* ... */;
    constexpr auto cs_position = /* ... */;

    return signature_list::component::make(
        cs_acceleration, cs_velocity, cs_position
        );
}

constexpr auto make_ssl()
{
    constexpr auto ss_acceleration = /* ... */;
    constexpr auto ss_velocity = /* ... */;
    constexpr auto ss_collision = /* ... */;

    return signature_list::system::make(
        ss_acceleration, ss_velocity, ss_collision
        );
}
```

7.2 Context settings

Context settings are mandatory options that need to be defined prior to `context` instantiation, implemented using option maps. The context needs to be aware of:

- All the previously defined *component signatures*, and *system signatures*. These will be passed to the settings as *signature lists*;
- The chosen **multithreading policy** and **system scheduling strategy**;
- The desired **entity storage policy**, which can be *dynamic* or *fixed*.
  - Entity and component insertion will be faster with a fixed limit, as no checks for possible reallocations are required.

Here is an example context settings definition and context instantiation:

```
constexpr auto context_settings =
    ecst::settings::make()
        .allow_inner_parallelism()
        .fixed_entity_limit(ecst::sz_v<10000>)
        .component_signatures(make_csl())
        .system_signatures(make_ssl());

auto context = ecst::context::make(context_settings);
```

Chapter 8

Execution flow

Using ECST requires the user to follow a particular **execution flow**, composed of different stages. The execution flow restricts possible operations *(using proxy objects)* in order to keep the state of the application consistent and to avoid a set of race conditions.

Before examining the execution flow, **critical operations** will be defined in the following section.

8.1 Critical operations

Some of the actions that can be executed in specific stages of the execution flow are called **critical operations**. These operations may require **memory allocations** and/or **synchronization** - they have to be executed sequentially. Execution of critical operations can happen immediately in some contexts *(e.g. during a **step**)* or can be delayed using **deferred functions** in other contexts *(e.g. system execution)*.

Critical operations include:

- Creating or destroying an entity;
- Adding or removing a component to/from an entity;
- Starting the execution of a system chain.

Non-critical operations include:

- Using a system's output data;
- Marking an entity as *"dead"*;
- Accessing or mutating existing component data.

An example containing critical and non-critical operations in a system implementation is shown below:

```
template <typename TData>
void process_collision_particles(TData& data)
{
    data.for_entities([&](auto eid)
        {
            // Reading/mutating components is a
            // non-critical operation.
            const auto& contact_list =
                data.get(ct::contact, eid);

            for(const auto& c : contact_list)
            {
                // Critical operation can be delayed
                // to a later stage.
                data.defer([&](auto& proxy)
                    {
                        // Creating entities and adding
                        // or removing components is a
                        // critical operation.
                        auto p = proxy.create_entity();
                        proxy.add_component(ct::particle, p);
                    });
            }

            if(!contact_list.empty())
            {
                // Marking an entity as "dead" is a
                // non-critical operation.
                data.kill_entity(eid);
            }
        });
}
```

8.2 Flow stages

ECST's execution flow is composed of the following stages: **step**, **system execution**, **refresh**.

8.2.1 Step

Step stages are accessed through a `context`, using a **step proxy**.

They:

- Allow immediate execution of *non-critical operations*;
- Allow immediate execution of *critical operations*;
- Allow execution of system chains;
- Execute a **refresh** after completion.

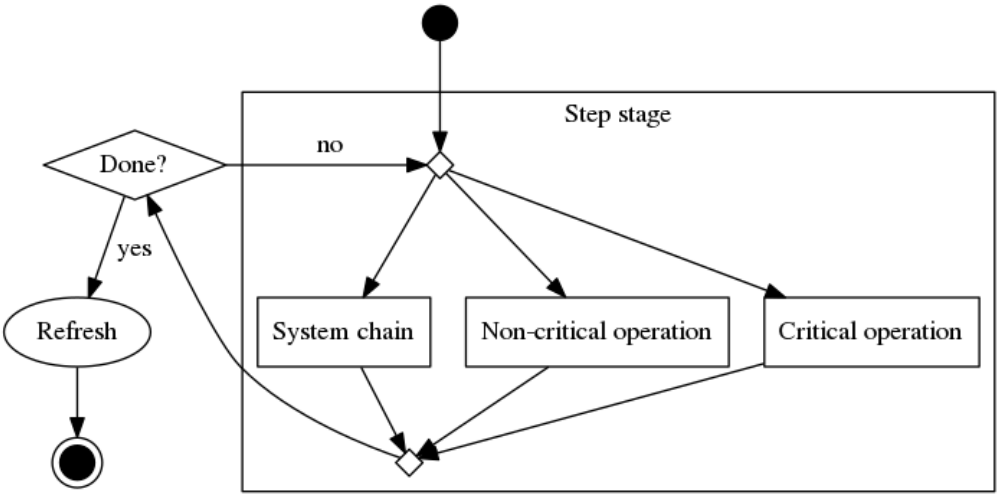


Figure 8.1: ECST flow: "step" stage overview

8.2.1.1 User code

A step can be defined in user code by calling `context::step(...)` with a function that accepts a *step proxy*. The following example shows a code snippet in which the user, inside of a step stage, prepares a rendering system, executes a system chain that processes physics and generates vertices, and finally renders the generated vertices to the window.

```
1  namespace sea = ::ecst::system_execution_adapter;
2
3  context.step([&](auto& proxy)
4  {
5      proxy.system(st::render).prepare();
```

```
6
7       proxy.execute_systems(
8           sea::t(st::physics).for_subtasks(
9               [dt](auto& s, auto& data)
10              {
11                  s.process(dt, data);
12              }),
13          sea::t(st::render).for_subtasks(
14              [](auto& s, auto& data)
15              {
16                  s.process(data);
17              })
18          );
19
20      proxy.for_system_outputs(st::render,
21          [&window](auto& s, auto& va)
22          {
23              window.draw(va.data(), va.size(),
24                  PrimitiveType::Triangles);
25          });
26  });
```

8.2.2 System execution

System execution stages are accessed through system implementations, using a **data proxy**.

They:

- Allow immediate execution of *non-critical operations*;
- Allow deferred execution of *critical operations*.

8.2.3 Refresh

Refresh stages are automatically executed after the completion of a *step stage*.

They sequentially perform the following operations:

1. Collects all deferred functions and executes them sequentially.
2. Reclaims all entities marked as “dead”, making the reuse of their IDs possible.
3. Matches all newly created, destroyed and modified entities to systems, subscribing or unsubscribing them.

A **refresh state** is instantiated at the beginning of the refresh, that will be used as a buffer in order to allow communication between all refresh stages. The refresh state contains a set of entity IDs to reclaim `to_kill` and a set of entity IDs to match `to_match`.

Algorithm 1: ECST flow: refresh algorithm overview

```
1 toKill ⟵ ∅;
2 toMatch ⟵ ∅;
3 ExecuteDeferredFunctions(toKill, toMatch) ;        // fills toKill and toMatch
4 ReclaimDeadEntities(toKill) ;                      // mutates and reads toKill
5 MatchEntitiesToSystems(toMatch) ;                  // reads toMatch
```

8.2.3.1 Deferred function execution

As deferred functions may contain *critical operations*, they need to be executed sequentially. Deferred functions are produced during system execution - every *subtask state* has its own deferred function list.

Algorithm 2: ECST flow: refresh - ExecuteDeferredFunctions

```
1 foreach instance i ∈ context c do
2     foreach state s ∈ instance i do
3         foreach function F ∈ state s do
              /* entity deletion mutats toKill                                  */
              /* entity creation or component addition/removal mutates toMatch
                 */
4             F(toKill, toMatch);
5         end
6     end
7 end
```

8.2.3.2 Dead entity reclamation

During system execution, entities may be marked as "dead" by subtasks. Every *subtask state* contains a *sparse set* of entity IDs marked as dead.

Algorithm 3: ECST flow: refresh - ReclaimDeadEntities

```
   /* add entities marked during system execution to toKill                */
 1 foreach instance i ∈ context c do
 2 |   foreach state s ∈ instance i do
 3 |   |   toKill ⟵ toKill ∪ s.toKill;
 4 |   end
 5 end
   /* unsubscribe dead entities from systems                               */
 6 foreach instance i ∈ context c do in parallel
 7 |   foreach entityID eid ∈ toKill do
 8 |   |   Unsubscribe(i, eid);
 9 |   end
10 end
   /* reclaim IDs for future use                                           */
11 foreach entityID eid ∈ toKill do
12 |   c.Reclaim(eid)
13 end
```

8.2.3.3 Entity-system matching

Newly created entities and entities with a mutated component set must be matched to systems. Checking if an entity matches a system is done by computing *dense bitset inclusion* between the entity's active component bitset and the system's required component bitset. Testing `std::bitset` inclusion is not part of the Standard Library *(although proposed, see [24])* - it can be implemented using bitwise operators as follows:

```
1  bool matches(component_bitset cb_entity, component_bitset cb_system)
2  {
3      return (cb_system & cb_entity) == cb_system;
4  }
```

A possible intuition for the code snippet above consists in thinking about systems as **keys** and entity instances as **locks**. If a key *fits in* a lock, the corresponding entity matches the system.

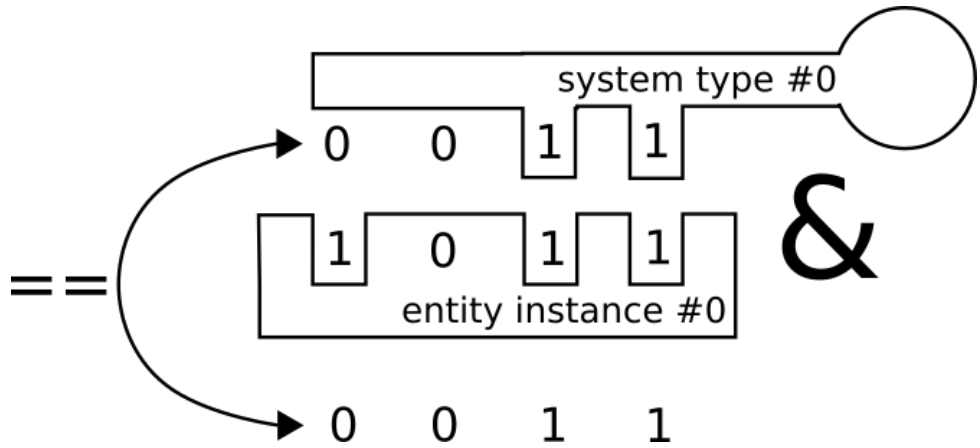


Figure 8.2: ECST flow: key/lock entity/system matching intuition

Bitwise inclusion tests can be executed in parallel over all system instances:

Algorithm 4: ECST flow: refresh - MatchEntitiesToSystems

```
1 foreach instance i ∈ context c do in parallel
2     foreach entityID eid ∈ toMatch do
3         if Matches(GetComponentBitset(i), GetComponentBitset(eid)) then
4             Subscribe(i, eid);
5         else
6             Unsubscribe(i, eid);
7         end
8     end
9 end
```

Chapter 9

Storage

ECST stores **entity metadata**, **component data**, and **system instances** inside the context object. To optimize performance depending on user-defined settings, *static dispatching (see Chapter 14, Section 14.3)* techniques are used. This chapter will cover the design and implementation of the aforementioned storage types.

9.1 Component data

Component data is stored in **chunks**. A **chunk** binds one or more component types to a particular **component storage strategy**. When the user tries to access a component by a tag `ct`, the chunk responsible for storing `ct` is found at compile-time, and the data is retrieved from the bound storage.

Chunks are stored inside the context in a `boost::hana::tuple`.

9.1.1 Component storage strategy

A **component storage strategy** is a class designed to store component data. Any class can be used as a storage strategy, as long as it satisfies the following requirements:

- It needs to provide two nested type names:
  - `component_tag_list_type`: a *type list* containing all the tags of components stored;

  - `metadata_type`: storage-specific metadata that will be *injected* in entity metadata. Can be used to store additional data required to retrieve components directly in the entities. Usually an empty class.

- It needs to implement the following interface:

```
// Given a component tag, an entity ID and its
// storage-specific metadata, returns a reference to
// the corresponding component.
template <typename TComponentTag, typename... Ts>
auto& get(TComponentTag, entity_id, const metadata_type&);

// Given a component tag, an entity ID and its
// storage-specific metadata, creates a component and
// returns a reference to it.
template <typename TComponentTag, typename... Ts>
auto& add(TComponentTag, entity_id, metadata_type&);
```

By default, the `contiguous_buffer`, `empty` and `hash_map` storage strategies are available.

- `contiguous_buffer` stores data in either an `std::array` or an `std::vector`, depending on whether or not the *entity limit* is fixed or dynamic.
- `empty` does not store any data - it's used for data-less components which only "mark" entities.
- `hash_map` stores data in an `std::unordered_map`. It should only be used for very big components with infrequent lookups/additions.

Users can create their own storage strategies by writing classes fulfilling the requirements mentioned above, and by providing a *"maker"* `struct` with the following interface:

```
template <typename TComponentTagList>
struct my_storage_strategy { /* ... */ };

struct my_storage_strategy_maker
{
    // Given context settings and a list of component tags, return
    // an `mp::type_c` wrapping the appropriate storage strategy.
    template <typename TSettings, typename TComponentTagList>
    constexpr auto make_type(TSettings, TComponentTagList) const

    {
```

```
12          // Static dispatching that depends on `TSettings` and
13          // `TComponentTagList` can be performed here to improve
14          // performance and memory usage.
15
16          return mp::type_c</* ... */>;
17      }
18  };
```

The newly created class can then be used during component signature definition:

```
1  constexpr auto cs_test_component =
2      cs::make(ct::test_component)
3          .storage_strategy(my_storage_strategy_maker{});
```

Several component storage strategy designs have been analyzed in depth by Adam Martin [25].

9.2 Entity metadata

Entity metadata can be accessed through entity IDs and deals with keeping track of active components, ensuring handle validity and storing *chunk metadata*.

It is implemented as a simple `struct`:

```
1  template <typename TComponentBitset, typename TChunkMetadataTuple>
2  struct metadata : TChunkMetadataTuple
3  {
4      TComponentBitset _bitset;
5      counter _counter;
6      // ...
7  };
```

`metadata` derives from `TChunkMetadataTuple` in order to take advantage of the **empty base optimization** *(EBO) (more details at [26])*. The stored `_counter` is incremented every time an entity IDs is reused - handles that try to access an entity check if their local counter matches with `_counter` to make sure they're not accessing another entity reusing the same ID [27].

All entity metadata instances, along with available entity IDs, are stored in either `ecst::entity::container::dynamic` or `ecst::entity::container::fixed`, depending on the entity limit specified by the user in context settings.

The dynamic storage uses an `std::vector` to store metadata and a *dynamic sparse integer set* to store available IDs. The fixed storage uses an `std::array` to store

metadata and a *fixed sparse integer set* to store available IDs - it is faster than the dynamic one as no checks for growth *(reallocation)* have to be performed.

Sparse integer sets are data structures extremely efficient for the management of entity IDs. They are analyzed in Chapter 14, Section 14.1.

9.3 Instances and systems

User-defined systems are stored inside **system instances**. Every system type has a corresponding system instance. All system instances are stored in the context, in an `std::tuple`, allowing users and other modules of the library to lookup instances at compile-time by system type.

9.3.1 Instance

A **system instance** is composed of the following members:

- An instance of the user-defined system type - systems can be stateful *(e.g. for caching reasons or to store a data structure)* and may require storage;
- A *sparse integer set* of the currently subscribed entity IDs;
- A *dense bitset* of the component types required for system subscription *(see Chapter 14, Section 14.2 for details)*;
- A **parallel executor** object that implements *inner parallelism*;
- A **state manager** object that binds a **state** to every *subtask*.

9.3.1.1 State

Every *subtask* has a corresponding **state**, which stores the following elements:

- *Output data* optionally generated from the subtask;
  - The data is set during subtask execution;
  - The data can be read from dependent systems or during a *step*.

- A `to_kill` sparse integer set, that keeps track of the entities marked as dead during subtask execution;
  - The set is filled during subtask execution;
  - The entities are reclaimed during a *refresh*.
- A `std::vector` of **deferred functions**.
  - The vector is filled during subtask execution;
  - The functions are sequentially executed during a *refresh*.

Chapter 10

Multithreading

Multithreading is used in ECST in the following situations, to potentially increase the run-time performance of user applications:

- **Outer parallelism**: system chains independent from each other can run in parallel. Implemented with **system scheduling**;
- **Inner parallelism**: system execution can be split over multiple *subtasks*. Implemented with **inner parallelism strategies** and **slicing**;
- **Refresh**: some operations in the refresh stage can run in parallel.

10.1 Thread pool

Execution of operations in separate threads is achieved through a simple **thread pool** which consists of a **lock-free queue** and a number of **workers**. Every worker runs on a separate thread and continuously dequeues **tasks** from the queue, executing them.

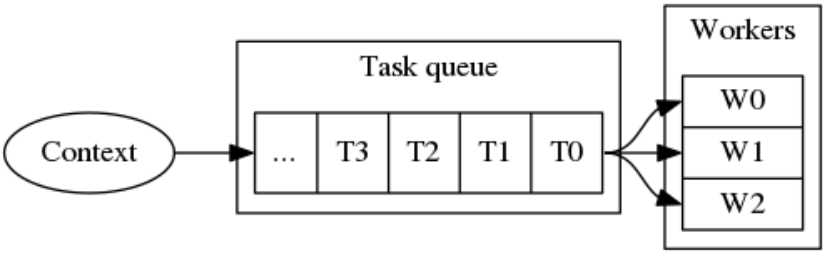


Figure 10.1: ECST multithreading: thread-pool architecture

Tasks are implemented using `ecst::fixed_function`, similar to `std::function` but with a fixed allocation size. Synchronization is implemented using `std::condition_variable` and simple `std::size_t` counters. An implementation making use of `std::packaged_task` and `std::future` was tested, but the unnecessary overhead brought by those classes was significant.

10.1.1 Lock-free queue

To provide **workers** with tasks, a third-party blocking concurrent lock-free queue developed by Cameron Desrochers is used. The queue class is called `moodycamel::BlockingConcurrentQueue`, and is available on GitHub under the *Simplified BSD License*.

The queue is used in the worker code as follows:

```
1  void worker::run()
2  {
3      task t;
4
5      while(_state == state::running)
6      {
7          _queue->wait_dequeue(t);
8          t();
9      }
10 }
```

The `_queue->wait_dequeue(t)` method call will block the current thread until the queue is not empty, and then dequeue a task into `t`. In order to prevent indefinitely waiting upon thread pool destruction, a number of *"dummy"* empty tasks are spawned to wake up the waiting workers:

```
thread_pool::~thread_pool()
{
    // Signal all workers to exit their processing loops.
    for(auto& w : _workers) w.stop();

    // Post dummy tasks until all workers have exited their loops.
    while(!all_workers_finished()) post_dummy_task();

    // Join the workers' threads.
    for(auto& w : _workers) w.join();
}
```

10.2 Synchronization

Synchronization and waiting are required when implementing both outer and inner parallelism:

- When executing outer parallelism, systems must wait for **all** their dependencies to complete before starting execution.
- When executing inner parallelism, **all** subtasks must be finished in order to complete a system execution.

The waiting conditions are very simple and can easily and efficiently be implemented using `std::condition_variable` in conjunction with a simple `std::size_t` counter. ECST provides a convenient and safe waiting interface, obtained by wrapping the aforementioned synchronization primitives alongside an `std::mutex` in a class called `counter_blocker`:

```
class counter_blocker
{
private:
    std::condition_variable _cv;
    std::mutex _mutex;
    std::size_t _counter;

public:
    counter_blocker(std::size_t initial_count);

    // Decrements the counter and notifies one waiting thread.
    void decrement_and_notify_one();

    // Decrements the counter and notifies all waiting threads.
    void decrement_and_notify_all();
```

```
16
17      // Executes `f` and blocks the caller until the counter
18      // reaches zero. Assumes that `f` will trigger a chain of
19      // operations that will decrement the counter.
20      template <typename TF>
21      void execute_and_wait_until_zero(TF&& f);
22  };
```

The `counter_blocker` can be used as follows:

```
1   // Create a `counter_blocker` initialized to `n`.
2   counter_blocker cb{n};
3
4   // Immediately execute the passed function and block
5   // until `cb` reaches zero.
6   cb.execute_and_wait_until_zero([&]
7       {
8           // `spawn_tasks` will decrement the counter.
9           spawn_tasks(cb, n);
10      });
```

Here's a possible implementation for `spawn_tasks`:

```
1   void spawn_tasks(counter_blocker& cb, int n)
2   {
3       for(int i = 0; i < n; ++i)
4       {
5           post_in_thread_pool([&]
6               {
7                   // Decrement the counter inside `cb` and
8                   // notify one thread.
9                   cb.decrement_and_notify_one();
10              });
11      }
12  }
```

The pattern shown above is used in the implementation of both outer and inner parallelism and in the refresh stage.

10.2.1 Implementation details

The synchronization operations are hidden behind interfaces that take references to the members of a `counter_blocker`. The public methods in `counter_blocker` call the following functions:

```
1   // Decrements `c` through `mutex`, and calls `cv.notify_one()`.
2   void decrement_cv_counter_and_notify_one(
```

```
    mutex_type& mutex, cv_type& cv, counter_type& c);

// Decrements `c` through `mutex`, and calls `cv.notify_all()`.
void decrement_cv_counter_and_notify_all(
    mutex_type& mutex, cv_type& cv, counter_type& c);

// Locks `mutex`, executes `f` and waits until `c` is zero
// through `cv`.
template <typename TF>
void execute_and_wait_until_counter_zero(
    mutex_type& mutex, cv_type& cv, counter_type& c, TF&& f);
```

The functions above call implementation functions to access the passed arguments. The most primitive implementation function is `access_cv_counter`, which calls a passed function after safely accessing the counter inside a `counter_blocker`:

```
template <typename TF>
void access_cv_counter(
    mutex_type& mutex, cv_type& cv, counter_type& c, TF&& f)
{
    lock_guard_type l(mutex);
    f(cv, c);
}
```

It is used as a building block for the *"decrement and notify"* functions:

```
void decrement_cv_counter_and_notify_one(
    mutex_type& m, cv_type& cv, counter_type& c)
{
    access_cv_counter(m, cv, c, [](auto& x_cv, auto& x_c)
        {
            assert(x_c > 0);
            --x_c;
            x_cv.notify_one();
        });
}
```

The *"execute and wait"* functions are implemented using an `std::unique_lock` waiting on a generic predicate through the `std::condition_variable` stored in the `counter_blocker`:

```
template <typename TPredicate, typename TF>
void execute_and_wait_until(
    mutex_type& m, cv_type& cv, TPredicate&& p, TF&& f)
{
    unique_lock_type l(m);
    f();
    cv.wait(l, p);
}
```

```
9
10  template <typename TF>
11  void execute_and_wait_until_counter_zero(
12      mutex_type& m, cv_type& cv, counter_type& c, TF&& f)
13  {
14      execute_and_wait_until(m, cv,
15          [&c]
16          {
17              return c == 0;
18          },
19          f);
20  }
```

10.3 System scheduling

System scheduling implements the concept of *outer parallelism*. The user can begin system execution during a *step stage* from any number of independent systems:

```
1   ctx.step([&](auto& proxy)
2   {
3       // Start execution of system chains from `s::s0` and `s::s1`:
4       proxy.execute_systems_from(st::s0, st::s1)(
5           sea::t(st::s0).for_subtasks([dt](auto& s, auto& data)
6               {
7                   // Overload for `s::s0`.
8                   s.process(dt, data);
9               }),
10          sea::t(st::s1).for_subtasks([dt](auto& s, auto& data)
11              {
12                  // Overload for `s::s1`.
13                  s.process(dt, data);
14              }));
15  });
```

The `s::s0` and `s::s1` system types need to be independent of each other *(a compile-time error will occur otherwise)*. After calling `proxy.execute_systems_from`, the system manager inside the context will instantiate a system scheduler and begin execution:

```
1   template <typename TSettings>
2   template <typename TCtx, typename TSystemTagList, typename... TFs>
3   void system_manager<TSettings>::execute_systems_impl(
4       TCtx& ctx, TSystemTagList sstl, TFs&&... fs)
5   {
6       // Instantiate system scheduler (specified in context settings).
7       scheduler_type s;
8
9       // Overload user-provided functions.
```

```
10      auto o_fs = boost::hana::overload_linearly(fs...);
11
12      // Begin execution.
13      // (Blocks until all systems in the chain have been executed.)
14      s.execute(ctx, sstl, os);
15  }
```

The current only default system scheduler type is called `atomic_counter` - it `counter_blocker` instances to wait for system dependencies' execution completion.

10.3.1 Atomic counter scheduler

The `atomic_counter` scheduler keeps count of the remaining systems *(yet to be executed)* and assigns a **task** to every system. Every task keeps count of its **remaining dependencies**. Tasks are executed recursively - after completing the starting independent tasks, every **dependent task** *(child task)* whose dependencies were satisfied is executed.

The scheduler will block until all systems have been executed, and all dependencies

between systems will be respected. Here's a high-level pseudocode of the algorithm:

Algorithm 5: ECST multithreading: system scheduler - atomic counter

```
   input  : List of starting system tags startSystems
   input  : Overloaded processing function f
 1 Algorithm ExecuteSystems(startSystems, f)
       /* count unique nodes traversed from every starting system      */
 2     remainingSystems ⟵ GetChainSize(startSystems);
       /* start recursive task execution                               */
 3     foreach system s ∈ startSystems do in parallel
 4         t ⟵ TaskFromSystem(s);
 5         RunTask(remainingSystems, t, f);
 6     end
       /* block until all systems have been executed                   */
 7     block thread while remainingSystems > 0;
 8 end

 9 Procedure RunTask(remainingSystems, t, f)
       /* execute current task                                         */
10     f(SystemOf(t));
11     DecrementAtomicCounter(remainingSystems);
       /* for each dependent child task                                */
12     foreach task dt ∈ DependentTasks(t) do
           /* notify dt the current (parent) task was executed         */
13         DecrementRemainingDependencies(dt);
           /* run dt recursively if its dependencies were satisfied    */
14         if RemainingDependenciesCount(dt) == 0 then
15             RunTask(remainingSystems, dt, f);
16         end
17     end
18 end
```

The implementation of the algorithm above will be now analyzed in the sections below.

10.3.1.1 Starting chain execution

The first step is traversing the implicit dependency **directed acyclic graph**, from every user-provided starting system type, counting the unique traversed nodes. This is done with a compile-time **breadth-first traversal**.

```
template <typename TCtx, typename TSystemTagList, typename TF>
void atomic_counter::execute(TCtx& ctx, TSystemTagList sstl, TF&& f)
{
    // Count of nodes traversed starting from every node in `sstl`.
    constexpr auto n = signature_list::system::chain_size(ssl, sstl);

    // Counter blocker used to block until all systems in the chain
    // have been executed.
    counter_blocker b{n};

    // Begin the execution and block until all systems have finished.
    b.execute_and_wait_until_zero([&]() mutable
        {
            this->start_execution(ctx, sstl, b, f);
        });
}
```

`atomic_counter::start_execution` will begin the recursive task execution.

10.3.1.2 Recursive task execution

Every system in the context has an assigned task, which has a unique ID. `atomic_counter::start_execution` retrieves the tasks of the starting systems and executes them recursively:

```
template <typename TCtx, typename TSystemTagList, typename TBlocker,
    typename TF>
void atomic_counter::start_execution(
    TCtx& ctx, TSystemTagList sstl, TBlocker& b, TF&& f)
{
    // For each system tag in `sstl`...
    boost::hana::for_each(sstl, [&](auto st) mutable
        {
            // Get the corresponding task ID.
            auto sid = sls::id_by_tag(this->ssl(), st);

            // Run task with ID `sid`.
            ctx.post_in_thread_pool([&]() mutable
                {
                    this->task_by_id(sid).run(b, id, sp, f);
                });
        });
}
```

After retrieving a task by ID, `atomic_counter::task::run` will effectively execute the overloaded user-provided processing function on the system and recursively run children tasks with no remaining dependencies:

```
template <typename TBlocker, typename TID, typename TCtx, typename TF>
void atomic_counter::task::run(TBlocker& b, TID sid, TCtx& ctx, TF&& f)
{
    // Get reference to system instance from task ID.
    auto& s_instance(ctx.instance_by_id(sid));

    // Execute overloaded processing function on system instance.
    s_instance.execute(ctx, f);

    // Safely decrement "remaining systems" counter.
    b.decrement_and_notify_one();

    // For every dependent task ID...
    for_dependent_ids([&](auto id)
        {
            // Retrieve the corresponding task.
            auto& dt = task_by_id(id);

            // Then, inform the task that one of its dependencies (the
            // current task) has been executed.
            dt.decrement_remaining_dependencies();

            if(dt.remaining_dependencies() == 0)
            {
                // Recursively run the dependent task.
                ctx.post_in_thread_pool([&]
                    {
                        dt.run(b, id, ctx, f);
                    });
            }
        });
}
```

10.4 Inner parallelism

Inner parallelism allows single systems to be run in parallel by *splitting* the range of their subscribed entities across a number of **subtasks**. Every subtask has its own **state** and can run in a separate thread. **Parallel executors**, which are obtained by composing **inner parallelism strategies**, implement the concept of *inner parallelism*.

10.4.1 Parallel executor

Every *system instance* stores a **parallel executor**. Parallel executors wrap inner parallelism strategies composed at compile-time by library users. Here is an example

definition of an inner parallelism strategy:

```
namespace ips = ecst::inner_parallelism::strategy;
namespace ipc = ecst::inner_parallelism::composer;

constexpr auto my_parallelism_strategy =
    ipc::none_below_threshold::v(sz_v<10000>,
        ips::split_evenly_fn::v_cores()
        );

constexpr auto ss_acceleration =
    ss::make(st::acceleration)
        .parallelism(my_parallelism_strategy)
        .read(ct::acceleration)
        .write(ct::velocity);
```

The code snippet above configures `s::acceleration` to run in a single thread if its subscriber count is less than 10000, otherwise it will be evenly split across the available CPU cores.

System instances invoke the parallel executor by passing a reference to the parent context and a **subtask adapter** function. The subtask adapter function for parallel execution takes the following arguments:

- **Split index**, which is the ID of the current subtask.
- Begin and end **slice indices**, which will be stored inside a data proxy. They are used to retrieve the target entity subset.

```
template <typename TContext, typename TF>
void instance</* ... */>::execute_in_parallel(TContext& ctx, TF&& f)
{
    // "Subtask adapter" lambda.
    auto st = [&](auto split_idx, auto i_begin, auto i_end)
    {
        // Create multi data proxy.
        auto dp = data_proxy::make_multi<TSystemSignature>(
            *this, ctx, split_idx, i_begin, i_end);

        // Execute the bound slice.
        f(dp);
    };

    _parallel_executor.execute(*this, ctx, std::move(st));
}
```

This design has been chosen in order to easily implement other system instance types in the future *(e.g. systems that directly process component data, without knowledge*

of entities). The parallel executor implementation will ask the caller instance to prepare execution of n subtasks:

```
template <typename TInstance, typename TCtx, typename TF>
void split_every_n</* ... */>::execute(
    TInstance& i, TCtx& ctx, TF&& f)
{
    // Perform strategy-related calculations.
    auto per_split = /* ... */;
    auto split_count = /* ... */;

    // Executes all subtasks. Blocks until completed.
    utils::prepare_execute_wait_subtasks(
       inst, ctx, split_count, per_split, f);
}
```

The `utils::prepare_execute_wait_subtasks` function takes care of calling the `instance::prepare_and_wait_subtasks` method, which initializes the `counter_blocker` with the number of produced subtasks and starts their execution. The method performs the following operations:

- It **clears** and **prepares** the *states* necessary for subtask execution.
- It instantiates a `counter_blocker` that will block until all subtasks have been executed.
- It creates an adapter *"run in separate thread"* function that will be used to run all subtasks except one in separate thread pool tasks. This will allow the current thread to execute one of the subtasks.

```
template <typename TContext, typename TF>
void instance</* ... */>::prepare_and_wait_n_subtasks(
    TContext& ctx, int n, TF&& f)
{
    // Prepare `n` states, but set the counter to `n - 1` since one
    // of the subtasks will be executed in the current thread.
    _sm.clear_and_prepare(n);
    counter_blocker b{n - 1};

    // Function accepting a callable object which will be executed
    // in a separate thread. Intended to be called from inner
    // parallelism strategy executors.
    auto run_in_separate_thread = [this, &ctx, &b](auto& xf)
    {
        return [this, &b, &ctx, &xf](auto&&... xs)
        {
```

```
17              ctx.post_in_thread_pool([&xf, &b, xs...]()
18                  {
19                      xf(xs...);
20                      b.decrement_and_notify_all();
21                  });
22          };
23      };
24
25      // Runs the parallel executor and waits until the remaining
26      // subtasks counter is zero.
27      b.execute_and_wait_until_zero([&f, &run_in_separate_thread]
28          {
29              f(run_in_separate_thread);
30          });
31  }
```

Chapter 11

Proxy objects

Proxy objects are used to provide the user with a *restricted interface* that increases the safety and readability of application code. Proxy objects are instantiated by ECST and passed as a reference in the following cases:

- During entity processing in systems, a **data proxy** object is used to access *component data* and *previous system outputs*.
  - Data proxies also give access to *deferred function* creation, which is done through **defer proxy** objects. Defer proxies allow the user to enqueue *critical operations* in a context where only non-critical operations can be executed.
- In order to immediately execute *critical operations* or begin *system chain execution*, a **step proxy** object must be used. Step proxies are created from the context and accessed through the `context::step` method.
  - **Executor proxy** objects are required to access *system processing functions* during system chain execution *(inside of a **step**)*. They are usually hidden behind more convenient abstractions provided by *system execution adapters*.

All proxies are implemented as template classes containing `private` callable objects *(and additional data)*, with a `public` interface that invokes the stored callables.

11.1 Data proxies

Data proxies are created during *inner parallelism slicing*. Every data proxy is bound to a particular *subtask state* and provides the following interface functions:

- Entity iteration:

```
// Iterates over entities assigned to the current subtask.
template <typename TF>
auto for_entities(TF&& f);

// Iterates over all entities in the system.
template <typename TF>
auto for_all_entities(TF&& f);

// Iterates over all entities not in the current subtask.
template <typename TF>
auto for_other_entities(TF&& f);
```

- Entity count:

```
// Count of entities of the current subtask.
auto entity_count() const;

// Count of all entities in the system.
auto all_entity_count() const;

// Count of entities not in the current subtask.
auto other_entity_count() const;
```

- Entity/component manipulation:

```
// Returns a reference to a component of `eid` with tag `ct`.
template <typename TComponentTag>
decltype(auto) get(TComponentTag ct, entity_id eid);

// Enqueues a "deferred function".
template <typename TF>
void defer(TF&& f);

// Marks an entity as "dead".
void kill_entity(entity_id eid);
```

- System output access:

```
// Returns a reference to the system's output data.
auto& output();

// Loops over the outputs of a previous system (dependency).
template <typename TSystemTag, typename TF>
decltype(auto) for_previous_outputs(TSystemTag st, TF&& f);
```

Data proxies begin their life during the definition of system processing overloads *(in a **step**)*:

```
ctx.step([&](auto& proxy)
    {
        proxy.execute_systems_from(st::s0, st::s1)(
            sea::t(st::s0).for_subtasks([](auto& s, auto& data)
                {
                    s.process(data);
                }),
            sea::t(st::s1).for_subtasks([](auto& s, auto& data)
                {
                    s.process(data);
                }));
    });
```

The `data` arguments shown above are created during *inner parallelism slicing* and automatically passed to the overloaded processing functions by the context. The system implementation can then access data proxies as follows:

```
struct s0
{
    template<typename TData>
    void process(TData& data)
    {
        data.for_entities([](auto eid){ /* ... */ });
    }
};
```

11.1.1 Defer proxies

Defer proxies are created by data proxies and can only be accessed through them. They provide an interface to enqueue critical operations that will be executed during a *refresh*:

- Entity/handle manipulation:

```
entity_id create_entity();
void kill_entity(entity_id);

handle create_handle(entity_id);
handle create_entity_and_handle();
auto valid_handle(const handle& h) const;
auto access(const handle&) const;
```

- Component access/manipulation:

```
template <typename TComponentTag>
decltype(auto) add_component(TComponentTag, entity_id);

template <typename TComponentTag>
decltype(auto) get_component(TComponentTag, entity_id);

template <typename TComponentTag>
void remove_component(TComponentTag, entity_id);
```

- System access:

```
template <typename TSystemTag>
auto& instance(TSystemTag);

template <typename TSystemTag>
auto& system(TSystemTag);

template <typename TSystemTag, typename TF>
decltype(auto) for_system_outputs(TSystemTag, TF&& f);
```

Here is an example of a defer proxy in use:

```
data.for_entities([&](auto eid)
    {
        data.defer([&](auto& proxy)
            {
                auto e = proxy.create_entity();
                proxy.add_component(ct::c0, e);
            });
    });
```

The enqueued operations will be executed at the end of a *step*, during the automatically-triggered refresh *"execute deferred functions"* phase.

11.2 Step proxies

Step proxies allow every operation that *defer proxies* do, in addition to functions which *begin system chain execution*:

```
// Executes all system chains starting from `sts...`.
template <typename... TStartSystemTags>
auto execute_systems_from(TStartSystemTags... sts);

// Executes all system chains.
auto execute_systems();
```

These functions are accessed through the `context::step` method, which creates a step proxy and passes it to an user-defined function. The method also accepts a variadic number of `fs_refresh...` refresh event handler functions: the feature will be covered in Chapter 12, Section 12.1.

```
template <typename TFStep, typename... TFsRefresh>
auto context::step(TFStep&& f_step, TFsRefresh&&... fs_refresh)
{
    auto refresh_event_handler =
        boost::hana::overload_linearly(fs_refresh...);

    // Ensure `refresh()` is automatically called after executing `f`.
    ECST_SCOPE_GUARD([this, reh = std::move(refresh_event_handler)]
        {
            this->refresh(std::move(reh));
        });

    // Clear refresh state.
    _refresh_state.clear();

    // Build context step proxy.
    step_proxy_type step_proxy{*this, _refresh_state};

    // Execute user-defined step.
    return f_step(step_proxy);
}
```

Here is an example usage of a step proxy:

```
ctx.step([&](auto& proxy)
    {
        proxy.execute_systems()(
            sea::all().for_subtasks([](auto& s, auto& data)
                {
                    s.process(data);
```

```
                });
    },
    ecst::refresh_event::on_unsubscribe(st::s0,
        [](s::s0& system, entity_id eid){ /* ... */ }));
```

11.2.1 Executor proxies

Executor proxies are created from *system instances* and used to execute system processing functions. They allow more fine-grained control over system execution and can only be accessed in a *step*, using the `.detailed` method of any *system execution adapter*. Here's a usage example:

```
ctx.step([](auto& proxy)
    {
        proxy.execute_systems()(
            sea::all().detailed([&](auto& system, auto& executor)
                {
                    // Code to run before execution.

                    executor.for_subtasks([&](auto& data)
                        {
                            // Code that runs in every subtask.
                            system.process(data);
                        });

                    // Code to run after execution.
                }));
    });
```

Normally, executor proxies are hidden behind the more convenient *(yet more limited) system execution adapter* `.for_subtasks` method interface.

Chapter 12

Advanced features

This chapter will cover the design and implementation of some less commonly used ECST features.

12.1 Refresh event handling

The refresh stage takes care of various important operations, such as reclaiming dead entity IDs and subscribing/unsubscribing entities to systems. These operations produce *events* that can be optionally handled by the user through **refresh event handling**.

When calling the `context::step` method, a variadic number of functions can be passed that will be overloaded to *catch* refresh events and execute code:

```
ctx.step([](auto& proxy){ /* ... */ },
    ecst::refresh_event::on_unsubscribe(st::s0,
        [](auto& instance, auto eid)
        {
            // Handle unsubscription of `eid` from `s::s0`.
        }),
    ecst::refresh_event::on_subscribe(st::s1,
        [](auto& instance, auto eid)
        {
            // Handle subscription of `eid` from `s::s1`.
        }),
    ecst::refresh_event::on_reclaim([](auto eid)
        {
            // Handle reclamation of `eid`.
        }));
```

12.1.1 Implementation details

Refresh event handling is completely optional and implemented with compile-time function overloading: it does not introduce any additional unnecessary run-time overhead:

```
template <typename TFStep, typename... TFsRefresh>
auto context::step(TFStep&& f_step, TFsRefresh&&... fs_refresh)
{
    // Creates the overload of refresh event handlers...
    auto refresh_event_handler =
        boost::hana::overload_linearly(fs_refresh...);

    ECST_SCOPE_GUARD([this, reh = std::move(refresh_event_handler)]
        {
            // ...and passes it to refresh.
            this->refresh(std::move(reh));
        });

    // ...
}
```

Every event has a unique type and `constexpr` variable:

```
namespace impl
{
    struct subscribed_t { };
    struct unsubscribed_t { };
    struct reclaimed_t { };

    constexpr subscribed_t subscribed{};
    constexpr unsubscribed_t unsubscribed{};
    constexpr reclaimed_t reclaimed{};
}
```

Refresh implementation stages invoke the created overloaded function using the types defined above, in order to trigger the correct overload. Here is an example:

```
void context::refresh_impl_kill_entities(
    TRefreshState& rs, TFRefresh&& f_refresh)
{
    // ...

    // Reclaim all dead entities and fire events.
    rs._to_kill.for_each([&](entity_id eid)
        {
            this->reclaim(eid);
            f_refresh(refresh_event::impl::reclaimed, eid);
```

```
        });

    // ...
}
```

The user interface functions return SFINAE-restricted lambdas that depend on the passed system tags:

```
namespace refresh_event
{
    template <typename TSystemTag, typename TF>
    auto on_subscribe(TSystemTag, TF&& f)
    {
        return [f](impl::subscribed_t, auto& inst, auto eid)
            ->impl::enable_matching_instance<
                decltype(inst), TSystemTag>
        {
            return f(inst, eid);
        };
    }
}
```

The restriction allows to have multiple overloads that differ on the passed system tag, and is implemented as follows:

```
template <typename TInstance, typename TSystemTag>
using enable_matching_instance =
    std::enable_if_t<check_tag<TInstance, TSystemTag>()>;
```

`impl::enable_matching_instance` is a type alias that makes use of `std::enable_if_t` *(see [28])*, which prevents functions from participating in **overload resolution** *(see [29])* if the `check_tag<TInstance, TSystemTag>()` expression evaluates to `false`. Here is the implementation of `check_tag`:

```
// Returns `true` if `TInstance` is the system instance with
// tag `TSystemTag`.
template <typename TInstance, typename TSystemTag>
constexpr auto check_tag()
{
    // Retrieve system type from instance.
    using system_type = typename decay_t<TInstance>::system_type;

    // Create tag from retrieved system type.
    constexpr auto system_tag = tag::system::v<system_type>;

    // Check type equality between created and passed tags.
    return std::is_same<
        decay_t<decltype(system_tag)>,
```

```
        decay_t<TSystemTag>
        >{};
}
```

12.2 System execution adapters

System execution adapters are used to define the *target systems* of user-defined *system processing functions* during a step. Users can match zero or more systems depending on their tags or custom `constexpr` predicate functions. All systems can also be conveniently matched. Here is an example of system execution adapters in use:

```
namespace sea = ecst::system_execution_adapter;

ctx.step([](auto& proxy)
    {
        proxy.execute_systems()(

            // Match systems `s::a` and `s::b`.
            sea::t(st::a, st::b)
                .for_subtasks([](auto& s, auto& data)
                    {
                        s.process_a(0, data);
                    }),

            // Match systems fulfilling `my_predicate`.
            sea::matching(my_predicate)
                .for_subtasks([](auto& s, auto& data)
                    {
                        s.process_b(1, data, 'a');
                    }),

            // Match remaining systems.
            sea::all()
                .for_subtasks([](auto& s, auto& data)
                    {
                        s.process_c(2, "test", data);
                    }));
    });
```

These constructs allow users to conveniently execute different processing functions *(that can have different interfaces)* on different systems.

12.2.1 Implementation details

The implementation of system execution adapters is conceptually similar to the one for refresh event handling: all interface functions in the `system_execution_adapter` namespace will return SFINAE-restricted functions that will be linearly overloaded by the `proxy.execute_systems` call. The resultant function will then be called with every system during system recursive task execution - only the matching overloads *(depending on tags or user-provided predicates)* will be invoked.

Every adapter is implemented using `system_execution_adapter::matching` as a building block:

- `system_execution_adapter::all` returns a `sea::matching` with a *tautology predicate*;
- `system_execution_adapter::t(...)` returns a `sea::matching` that uses a predicate very similar to the previously analyzed `check_tag` function to restrict systems depending on their tags.

12.3 Entity handles

In order to **track** *particular entity instances*, users can create and manage **handles**. Handles are lightweight copyable objects *(usually as big as two pointers)* that can be used as parameters for functions provided by defer proxies, step proxies and by the context object:

```
1  // Returns `true` if `h` is not a "null handle".
2  auto valid_handle(const handle& h) const;
3
4  // Returns the entity ID of the entity pointed by `h`.
5  // Asserts `valid_handle(h)`.
6  auto access(const handle&) const;
7
8  // Returns `true` if the entity pointed by `h` is alive.
9  // Asserts `valid_handle(h)`.
10 auto alive(const handle& h) const;
```

Handles can be used to safely check whether or not an entity was destroyed, even if its ID has been reused:

```
auto e0 = context.create_entity();
auto h = context.create_handle(e);

// ...

if(context.alive(h))
{
    auto e1 = context.access(h);
}
```

12.3.1 Implementation details

Handles are simple structs with two fields: the entity ID they are pointing to and a validity counter.

```
struct handle
{
    entity_id _id;
    counter _ctr;
};
```

A special entity ID, equal to maximum finite value representable by the underlying numeric type, is the `invalid_id`. Handles pointing to `invalid_id` are considered **null** or **invalid** handles - `valid_handle` returns `false` for them.

Accessing entity metadata through an handle consists of the following steps:

- Check if the handle's counter is equal to the counter in the entity storage.
  - If the counter is not equal, the entity ID has been re-used and the handle points to a "dead" entity.
  - If the counter is equal, the corresponding metadata can be accessed through the handle's stored ID.

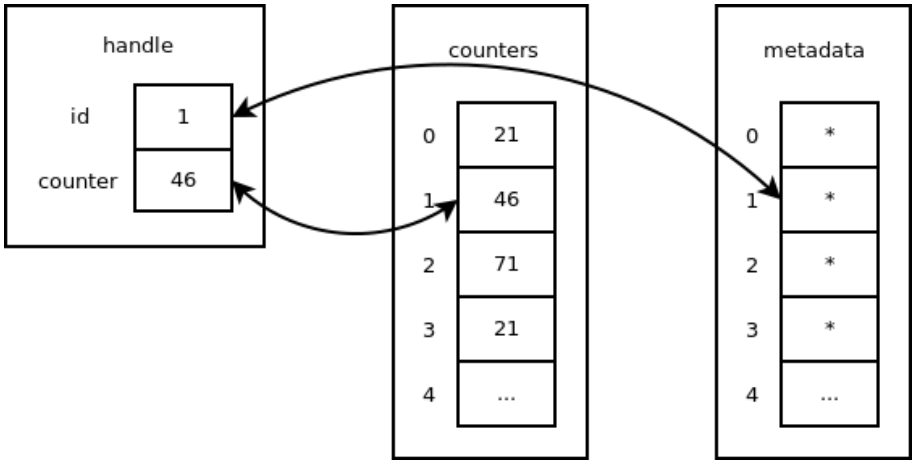


Figure 12.1: ECST advanced features: accessing entity metadata through handle

Chapter 13

Future work

ECST is in an experimental state and several design and implementation choices are subject to change. This chapter aims to provide a brief overview of the currently planned additions and adjustments.

13.1 System instance generalization

Currently *system instances* are closely tied to the concept of entity subscription and unsubscription. In the future, a more general system instance class will be introduced that will represent a **"generic computation step"** in the depedency DAG.

Generalized system instances will have no knowledge of entities and purely act on component data: this concept, alongside specialized component storage strategies, will allow to easily deal with SIMD operations.

13.2 Deferred function queue

Deferred functions are implemented using `std::vector<std::function>` in the current version of ECST, introducing unnecessary run-time overhead due to dynamic allocation and polymorphism.

As suggested by Matt Calabrese, a specialized **"deferred function queue"** class will be introduced in the future, which will store callable objects of various size in

the same resizable buffer. This will be achieved using fixed-size structures that will act similarly to *"vtables"*, which will be stored before their corresponding callable object in the buffer. A separate lightweight index will keep track of all the vtables - iterating over them will allow efficient sequential execution of deferred functions.

Appending functions to the queue will not require any additional memory allocation.

13.3 Declarative option map interfaces

Definining interfaces for *option maps* requires the creation of a class and multiple methods which explicitly need to check option value domains and update the option map. The possibility of streamlining the definitions of compile-time option interfaces using a declarative approach will be explored in the future - the goal is to create a small code-generation library/module that will generate rich compile-time option sets with an intuitive and safe user interface.

13.4 Streaming system outputs

Currently, a system depedendent on the output of another must wait until its parent's execution has been completed. Sometimes outputs could be processed as soon as they are generated connecting systems through a **streaming queue** - the inclusion of this feature will be investigated in the future.

Chapter 14

Miscellaneous

14.1 Sparse integer sets

A **sparse integer set** is a data structures representing a set of positive integers with the following time complexity characteristics:

	Time complexity
Check presence of integer	$\mathcal{O}(1)$
Add integer to set	$\mathcal{O}(1)$
Remove integer from set	$\mathcal{O}(1)$
Iterate over integers	$\mathcal{O}(n)$

Its space complexity is $\mathcal{O}(n)$.

Sparse integers sets are perfect for entity ID management, as the complexity of all required operations is optimal. They have been extensively covered in [30] and [31]. A possible C++ implementation is analyzed in [32]. Sparse integer sets have been already used in existing ECS libraries: an example is *Diana*, by Vincent Adam Burns [33].

14.1.1 Implementation details

Sparse integer sets are implemented in **vrm_core** *(see [34])*, which is a dependency of ECST. The used storage types depend on context settings *(static dispatching)*,

but conceptually every implementation is composed of the following elements:

- An unordered **dense array** D, which contiguously stores all integers present in the set;
- A **sparse array** S, containing indices to the elements of D, or special null $\emptyset$ values.

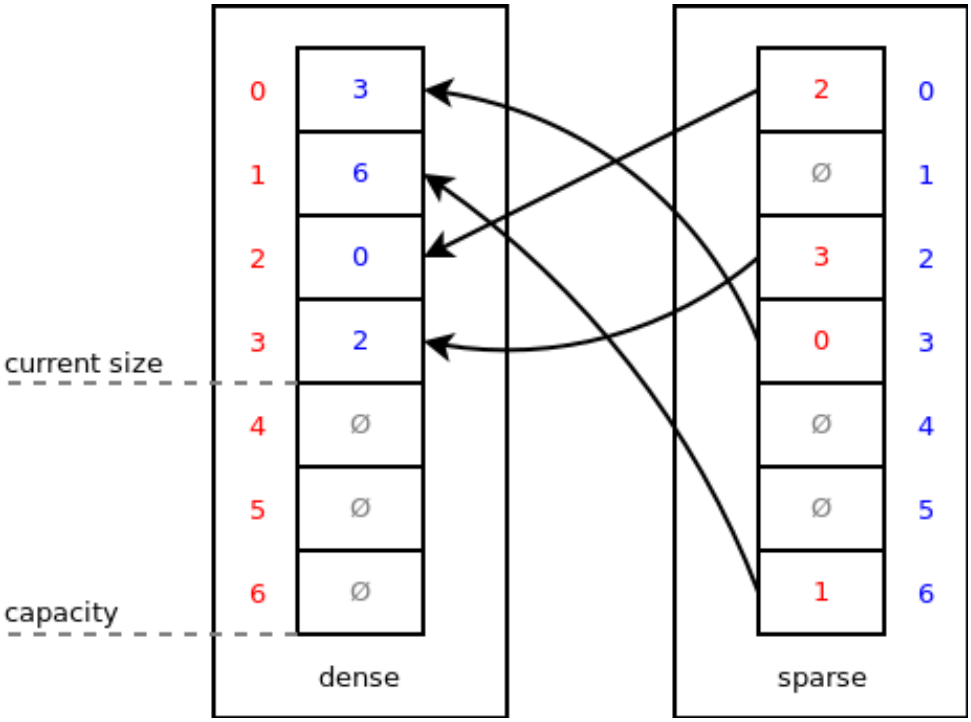


Figure 14.1: ECST miscellaneous: fixed sparse integer set example

14.1.1.1 Operation: contains

Checking whether or not an integer i is in the set consists in checking if $S_i \neq \emptyset$.

14.1.1.2 Operation: iteration

Iterating over the integers in the set consists in iterating over D.

14.1.1.3 Operation: add integer to set

Adding an integer i to the set consists in appending it to D and making S_i *point* to D_{last}.

Algorithm 6: ECST miscellaneous: SparseIntSet - AddInteger

```
   /* do nothing if i is already in the set                    */
1  if Contains(i) = false then
       /* append i to D                                        */
2      D[last] ⟵ i;
       /* make S[i] "point" to i                               */
3      S[i] ⟵ last;
4  end
```

14.1.1.4 Operation: remove integer from set

Removing an integer i from the set consists in swapping i with D_{last} if necessary, then updating S_i and decrementing the number of contained integers.

Algorithm 7: ECST miscellaneous: SparseIntSet - RemoveInteger

```
   /* do nothing if i is not in the set                        */
1  if Contains(i) = true then
       /* access S[i]                                          */
2      iPtr ⟵ S[i];
       /* check if iPtr is the last element in D               */
3      if D[last] ≠ D[iPtr] then
           /* if not, swap iPtr with the last element and update S */
4          D[iPtr] ⟵ D[last];
5          S[last] ⟵ iPtr;
6      end
       /* nullify S[i] and decrement last                      */
7      S[i] ⟵ ∅;
8      −−last;
9  end
```

14.2 Component bitset creation

Component bitsets are *dense bitsets* implemented with `std::bitset` used to keep track of an entity's components and to efficiently check whether or not an entity belongs to a system. Every component type is represented by a unique bit.

System instances store a component bitset that represents the required component types for subscription. The bitset is generated from a system signature, using compile-time tuple iteration and Boost.Hana algorithms.

Firstly, the component types specified in the system signature are retrieved and concatenated:

```
template <typename TSystemSignature, typename TSettings>
auto make_from_system_signature(TSystemSignature ss, TSettings s)
{
    // The "read" and "write" component tags are concatenated.
    auto all = boost::hana::concat(
        ss.read_ctag_list(), ss.write_ctag_list());

    // The complete list is passed to the `make_from_tag_list`
    // function, which will create the bitset.
    return make_from_tag_list(s, all);
}
```

Tags are values, and that type lists are `boost::hana::tuple` instances. Boost.Hana provides a `boost::hana::for_each` function that can be used to *"iterate"* over the elements of a tuple at compile-time. Building the bitset simply consists in iterating over `ctag_list` and setting the corresponding component bits to `1`:

```
template <typename TSettings, typename TComponentTagList>
auto make_from_tag_list(TSettings s, TComponentTagList ctl)
{
    // Create empty `std::bitset` with length equal to the
    // number of components.
    component_bitset<TSettings> b;

    // Retrieve the complete list of component signatures
    // from the context settings.
    auto csl = settings::component_signature_list(s);

    // For each tag in `ctl`...
    boost::hana::for_each(ctl, [&](auto ct)
        {
            // Retrieve the unique component ID from `csl`.
            auto id(signature_list::component::id_by_tag(csl, ct));

            // Set the corresponding bit.
            b.set(id, true);
        });

    return b;
}
```

14.3 Static dispatching

"Static dispatching" is the term used in ECST to refer to compile-time choices regarding data structures. It is used to avoid unnecessary overhead depending on user-specified context settings. An example use case can be found in the implementation of entity metadata storage: `std::array` or `std::vector` will be used to store metadata depending on the *fixed*/*dynamic* entity limit choice made by the user.

14.3.1 Implementation details

Most occurrences of static dispatching are implemented using `static_if`, available in **vrm_core** *(see [34])*. A compile-time branching construct is not yet part of the standard, but it has been proposed several times *(see [35], [36], [37] and [38])* and it's likely to be introduced in C++17. Nevertheless, a `static_if` construct that's more convienient and localized than *explicit template specialization* [39] can be implemented using C++14 features *(see [40], [41] and [42])*.

Using `static_if`, implementing static dispatching becomes straightforward. An auxiliary `dispatch_on_storage_type` function executes one of the passed callable objects depending on the user-specified storage limitations:

```
1  template <typename TSettings, typename TFFixed, typename TFDynamic>
2  auto dispatch_on_storage_type(
3      TSettings&& s, TFFixed&& f_fixed, TFDynamic&& f_dynamic)
4  {
5      return static_if(s.has_fixed_capacity())
6          .then([&](auto xs)
7              {
8                  return f_fixed(xs.get_fixed_capacity());
9              })
10         .else_([&](auto xs)
11             {
12                 return f_dynamic(xs.get_dynamic_capacity());
13             })(s);
14 }
```

Using the function above, data structures wrapped in `boost::hana::type_c` can be returned and later instantiated. Here is the implementation of the function statically dispatching entity metadata storage types:

```
template <typename TSettings>
auto dispatch_entity_storage(TSettings s)
{
    return settings::dispatch_on_storage_type(s,
        [](auto fixed_capacity)
        {
            return boost::hana::type_c<
                impl::fixed_entity_storage<
                    metadata_type<TSettings>,
                    fixed_capacity>
                >;
        },
        [](auto)
        {
            return boost::hana::type_c<
                impl::dynamic_entity_storage<
                    TSettings,
                    metadata_type<TSettings>>
                >;
        });
}
```

14.4 Compile-time breadth-first traversal

A compile-time version of the **breadth-first traversal** algorithm was implemented in order to allow users to begin system execution from particular independent nodes. A complete system signature list represents a DAG that can be composed of multiple *connected components* - knowledge of the exact number of nodes in a connected component is required to properly execute the system chain using the "atomic counter" scheduler.

Graph traversal algorithms provide a straightforward way of counting the number of unique nodes in a connected component. The BFT algorithm was chosen for this task: in short, it traverses a graph starting from a particular node and exploring all neighbor nodes first. Explored nodes have to be *"marked"* to prevent redundant traversals.

Breadth-first traversal is easy to implement using mutable data structures:

- A queue is used to keep track of the nodes that need to be explored;
  - Unmarked neighbors of the node currently being explored are enqueued for future traversal.

- Explored nodes need to be marked as *"visited"* to guarantee each node being traversed exactly once.

Similarly to the implementation of option maps, the required state is implemented using immutable Boost.Hana data structures whose operations yield a new *(copy)* updated structure instead of mutating the structure in-place.

- Nodes are compile-time numerical IDs *(`boost::hana::integral_constant`)*;
- The BFT queue is a `boost::hana::tuple`;
- Nodes are *"marked as visited"* by storing their ID in a `boost::hana::tuple`;
- Both the queue and the *"visited nodes tuple"* are stored in a `boost::hana::pair`. The pair is referred to as the **BFT context**.

All operations are defined inside the `bf_traversal` namespace. The data structures can be initialized and accessed with the following functions:

```
1  namespace bf_traversal
2  {
3      template <typename TStartNodeList>
4      constexpr auto make(TStartNodeList&& snl)
5      {
6          return hana::make_pair(snl, hana::make_tuple());
7      }
8
9      template <typename TBFTContext>
10     constexpr auto queue(TBFTContext&& c)
11     {
12         return hana::first(c);
13     }
14
15     template <typename TBFTContext>
16     constexpr auto visited(TBFTContext&& c)
17     {
18         return hana::second(c);
19     }
20 }
```

Convenient functions to query the state of the BFT context are defined as well:

```
1  namespace bf_traversal
2  {
3      template <typename TBFTContext, typename TNode>
4      constexpr auto is_visited(TBFTContext&& c, TNode&& n)
5      {
```

```
6           return hana::contains(visited(c), n);
7       }
8
9       template <typename TBFTContext, typename TNode>
10      constexpr auto is_in_queue(TBFTContext&& c, TNode&& n)
11      {
12          return hana::contains(queue(c), n);
13      }
14
15      template <typename TBFTContext>
16      constexpr auto is_queue_empty(TBFTContext&& c)
17      {
18          return hana::is_empty(queue(c));
19      }
20
21      template <typename TBFTContext>
22      constexpr auto top_node(TBFTContext&& c)
23      {
24          return hana::front(queue(c));
25      }
26  }
```

The algorithm is executed through the `bf_traversal::execute` function, which takes a list of starting nodes and the complete system signature list as parameters. A `step` lambda, which repeatedly yields updated BFT context instances, is executed recursively until the traversal is completed by using `boost::hana::fix`, which is an implementation of the **"Y-combinator"** *(a.k.a. "fixed-point combinator")*. `bf_traversal::execute` returns a list of the traversed node IDs.

```
1   template <typename TStartNodeList, typename TSSL>
2   auto execute(TStartNodeList&& snl, TSSL ssl)
3   {
4       // Recursive step.
5       // Takes itself as `self`, and the current context.
6       auto step = [=](auto self, auto&& ctx)
7       {
8           return static_if(bf_traversal::is_queue_empty(ctx))
9               .then([=](auto)
10                  {
11                      // Base case: empty BFT queue.
12                      // Return an empty tuple.
13                      return hana::make_tuple();
14                  })
15              .else_([=](auto&& x_ctx)
16                  {
17                      // Recursive case.
18                      // Call `self` with the updated context
19                      // returned by `step_forward`.
20                      // Append the last visited node to the
21                      // final result.
```

```
22
23                      auto updated_ctx =
24                          step_forward(x_ctx, ssl);
25
26                      return hana::append(
27                          self(updated_ctx),
28                          top_node(x_ctx));
29                  })(ctx);
30      };
31
32      // Begin the recursion vy initializing a BFT context.
33      return hana::fix(step)(make(snl));
34  }
```

The core of the algorithm resides in the `bf_traversal::step_forward` function:

```
1   template <typename TBFTContext, typename TSSL>
2   auto step_forward(TBFTContext&& c, TSSL ssl) noexcept
3   {
4       // Dequeue the first node.
5       auto pop_queue = hana::remove_at(queue(c), mp::sz_v<0>);
6
7       // List of neighbors of the dequeued node.
8       auto neighbors = dependent_ids_list(
9           ssl, signature_by_id(ssl, top_node(c)));
10
11      // Filter out already visited neighbors.
12      auto unvisited_neighbors =
13          hana::remove_if(neighbors, [=](auto x_nbr)
14              {
15                  return is_visited(c, x_nbr);
16              });
17
18      // Updated queue.
19      auto new_queue =
20          hana::concat(pop_queue, unvisited_neighbors);
21
22      // Updated "visited nodes" list.
23      auto new_visited =
24          hana::concat(visited(c), unvisited_neighbors);
25
26      // Updated BFT context.
27      return hana::make_pair(new_queue, new_visited);
28  }
```

The last missing piece is the implementation of `dependent_ids_list`, which returns a list of nodes that depend on the passed *"parent node"*. It is implemented by iterating over the complete system signature list *(using `boost::hana::fold_right`)*, gathering all nodes that have the *"parent node"* as one of their dependencies:

```
template <typename TSystemSignatureList, typename TSystemSignature>
auto dependent_ids_list(
    TSystemSignatureList ssl, TSystemSignature parent)
{
    namespace ss = signature::system;
    namespace sls = signature_list::system;

    // Retrieve the id of `parent`.
    auto parent_id = sls::id_by_signature(ssl, parent);

    // Build a list of dependent IDs.
    return hana::fold_right(ssl, hana::make_tuple(),
        [=](auto ss, auto acc)
        {
            // Check if `parent_id` is one of `ss`'s depedendencies.
            auto dl = sls::dependencies_as_id_list(ssl, ss);
            return static_if(hana::contains(dl, parent_id))
                .then([=](auto x_acc)
                    {
                        // If so, add `ss`'s ID to the result list.
                        auto ss_id = sls::id_by_signature(ssl, ss);
                        return hana::append(x_acc, ss_id);
                    })
                .else_([=](auto x_acc)
                    {
                        return x_acc;
                    })(acc);
        });
}
```

The algorithm is used in the implementation of the `atomic_counter::execute` scheduler function, under the name `chain_size`, in order to instantiate a `counter_blocker` that will block the calling thread until all systems belonging to a particular connected component of the dependency DAG have been executed.

Part III

Example ECST applications and benchmarks

Chapter 15

Overview

In this part two simple applications implemented using ECST will be analyzed, in order to provide the readers with usage examples of the library and with performance comparisons between various combinations of the compile-time multithreading options:

- A **2D real-time particle simulation**, where circular particles collide with each other in a closed space;
- A **real-time entity creation/destruction benchmark**, where entities with a limited life replicate themselves a fixed amout of times.

Chapter 16

2D particle simulation

16.1 Description

The simulation consists in a number of **rigid-body circular particles** of various radius, colliding with each other in a closed space. The particle entities are generated at the beginning of the demo. A **2D grid** spatial partitioning system is used to speed-up broadphase collision detection. Every particle has a **life** timer that constantly gets decremented: the entity is destroyed as soon as the timer reaches zero. When all the particles are destroyed, the simulation automatically ends.

Multiple simulations are executed and benchmarked, combining the following compile-time options and parameters:

- Entity count: 50000, 100000 and 200000;
- Inner parallelism: **enabled** or **disabled**;
- Entity storage strategy: **fixed** or **dynamic**.

In total, 12 simulations are executed.

The SFML library is used for rendering and math utilities.

16.2 Components

Every particle is composed of the following component types:

- **Position**: 2D `float` vector;

```
1    struct position { sf::Vector2f _v; };
```

- **Velocity**: 2D `float` vector;

```
1    struct velocity { sf::Vector2f _v; };
```

- **Acceleration**: 2D `float` vector;

```
1    struct acceleration { sf::Vector2f _v; };
```

- **Color**: used for SFML rendering;

```
1    struct color { sf::Color _v; };
```

- **Circle**: shape of the particle, controls its radius;

```
1    struct circle { float _radius; };
```

- **Life**: controls the lifetime of a particle.

```
1    struct life { float _v; };
```

16.3 Systems

- **Acceleration**: increments each particle's `velocity` vector by its `acceleration` vector;
  - Multithreading is enabled.
- **Velocity**: increments each particle's `position` vector by its `velocity` vector;
  - Multithreading is enabled.
- **Keep in bounds**: keeps every particle inside the boundaries of the simulation;
  - Multithreading is enabled.
- **Spatial partition**: stores the 2D spatial partitioning grid and manages it;

  - Multithreading is enabled;
  - Produces `std::vector<sp_data>` outputs. `sp_data` is a lightweight `struct` holding an `entity_id` and a pair of 2D cells coordinates. The produced vectors will be read from the context step in order to fill the stored 2D grid data structure.

- **Collision detection**: detects collisions between particles *(filtered by the broad-phase spartial partitioning system)*. The detected collisions are resolved by a subsequent system;
  - Multithreading is enabled;
  - Produces `std::vector<contact>` outputs. `contact` is a lightweight `struct` holding `entity_id` instances of two colliding particles. The contacts sequentially read from the `solve_contacts` system to solve penetration between particles.

- **Solve contacts**: reads `collision`'s produced `contact` outputs and sequentially solves penetration between particles;
  - Multithreading is disabled.

- **Render colored circle**: deals with particle rendering;
  - Multithreading is enabled;
  - Produces `std::vector<sf::Vertex>` outputs. The vertices are used to render the circles using SFML.

- **Life**: deals with particle lifetime. Continuously decreases every particle's `life` value and marks particles as dead when their lifetime is over;
  - Multithreading is enabled.

- **Fade**: changes particles' opacity depending on their life.
  - Multithreading is enabled.

The implicitly generated depedency DAG is shown below:

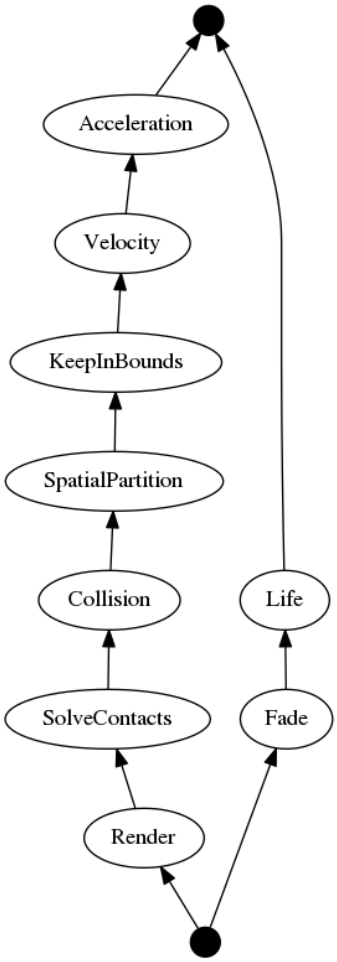


Figure 16.1: Particle simulation: dependency DAG

16.3.1 System implementations

Most of the system implementations are straightforward: `s::acceleration` and `s::velocity` simply iterate over their subscribed entities, mutating the values of the targeted components. Here is the implementation of `s::acceleration`:

```
1  struct acceleration
2  {
3      template <typename TData>
4      void process(ft dt, TData& data)
5      {
6          data.for_entities([&](auto eid)
7              {
```

```
                auto& v = data.get(ct::velocity, eid)._v;
                const auto& a = data.get(ct::acceleration, eid)._v;
                v += a * dt;
            });
    }
};
```

`s::fade`, `s::life` and `s::keep_in_bounds` are implemented in a similar way.

16.3.1.1 Spatial partition

`s::spatial_parititon` is a stateful system: it stores a 2D grid used to speed-up broadphase collisions and produces outputs later read in *step stage* to mutate the stored grid.

The grid is implemented with two nested `std::array` of `std::vector<entity_id>`:

```
struct spatial_partition
{
    using cell_type = std::vector<ecst::entity_id>;
    std::array<std::array<cell_type, grid_height>, grid_width> _grid;
    // ...
```

At the beginning of every step stage `_grid` is cleared. The `process` method of the system iterates over the subscribed entities and produces `sp_data` instances:

```
template <typename TData>
void process(TData& data)
{
    // Get a reference to the output vector and clear it.
    auto& o = data.output();
    o.clear();

    // For every entity in the subtask...
    data.for_entities([&](auto eid)
        {
            // Access component data.
            const auto& p = data.get(ct::position, eid)._v;
            const auto& c = data.get(ct::circle, eid)._radius;

            // Figure out the broadphase cell and emplace an
            // `sp_data` instance in the output vector.
            this->for_cells_of(p, c, [eid, &o](auto cx, auto cy)
                {
                    o.emplace_back(eid, cx, cy);
                });
        });
}
```

The `sp_data` struct is defined as follows:

```
struct sp_data
{
    ecst::entity_id _e;
    sz_t _cell_x, _cell_y;
};
```

Every subtask of `s::spatial_partition` will produce `sp_data` instances in parallel. They will be sequentially read in the step stage to fill the 2D grid:

```
sea::t(st::spatial_partition).detailed_instance(
    [&proxy](auto& instance, auto& executor)
    {
        // Clear 2D grid.
        auto& s(instance.system());
        s.clear_cells();

        // Produce `sp_data` instances in parallel.
        executor.for_subtasks([&s](auto& data)
            {
                s.process(data);
            });

        // Fill 2D grid sequentially.
        instance.for_outputs(
            [](auto& xs, auto& sp_vector)
            {
                for(const auto& x : sp_vector)
                {
                    xs.add_sp(x);
                }
            });
    }));
```

16.3.1.2 Collision

The `s::collision` system will iterate over unique pairs of particles in the same spatial partitioning cell and produce `contact` instances in parallel that will be sequentially processed by the `s::solve_contacts` system.

The `contact` struct is defined as follows:

```
struct contact
{
    // IDs of the colliding entities.
    ecst::entity_id _e0, _e1;
```

```
5
6       // Distance between entities.
7       float _dist;
8   };
```

As `s::spatial_parititon` is a dependency of `s::collision`, its state can be safely accessed in `s::collision::process`:

```
1   template <typename TData>
2   void process(TData& data)
3   {
4       // Get a reference to the output vector and clear it.
5       auto& out = data.output();
6       out.clear();
7
8       // Get a reference to the `spatial_partition` system.
9       auto& sp = data.system(st::spatial_partition);
10
11      // For every entity in the subtask...
12      data.for_entities([&](auto eid)
13          {
14              // Access the grid cell containing position `p0`.
15              auto& p0 = data.get(ct::position, eid)._v;
16              auto& cell = sp.cell_by_pos(p0);
17
18              for_unique_pairs(cell, eid, [&](auto eid2)
19                  {
20                      // Check "circle vs circle" collision
21                      // and eventually emplace a `contact`
22                      // instance in `out`.
23                  });
24          });
25  }
```

16.4 Results

16.4.1 Screenshots

Two screenshots of the particle simulations are shown below. The first one shows 50000 particle entities colliding in a closed space:

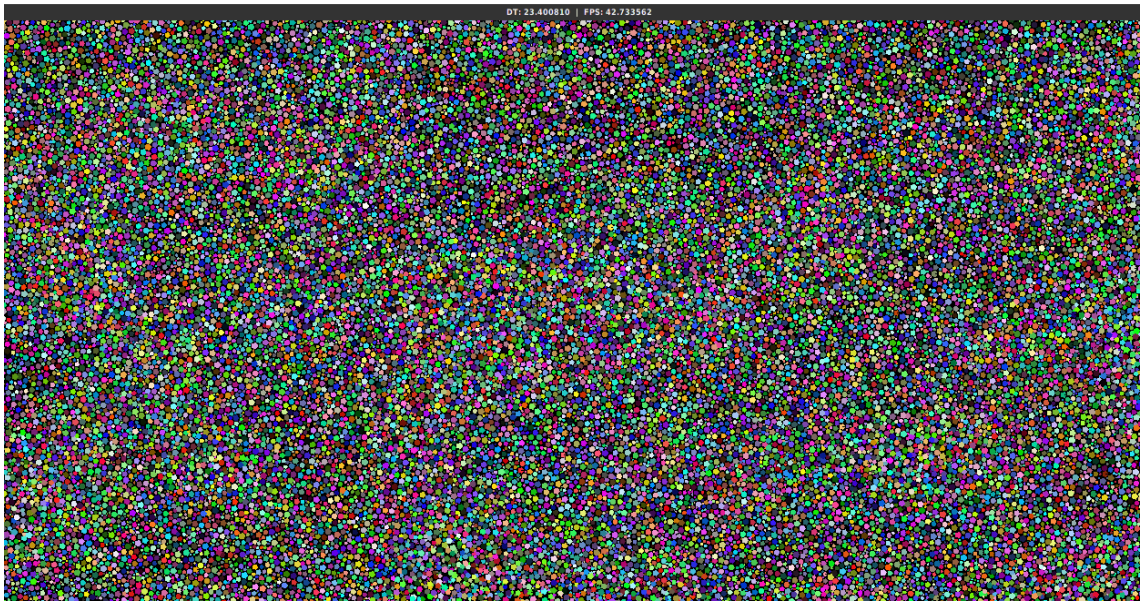


Figure 16.2: Particle simulation: screenshot - 50000 colliding particles

The second one shows one of the cells of the spatial partitioning 2D grid - all particles belonging to the cell are highlighted:

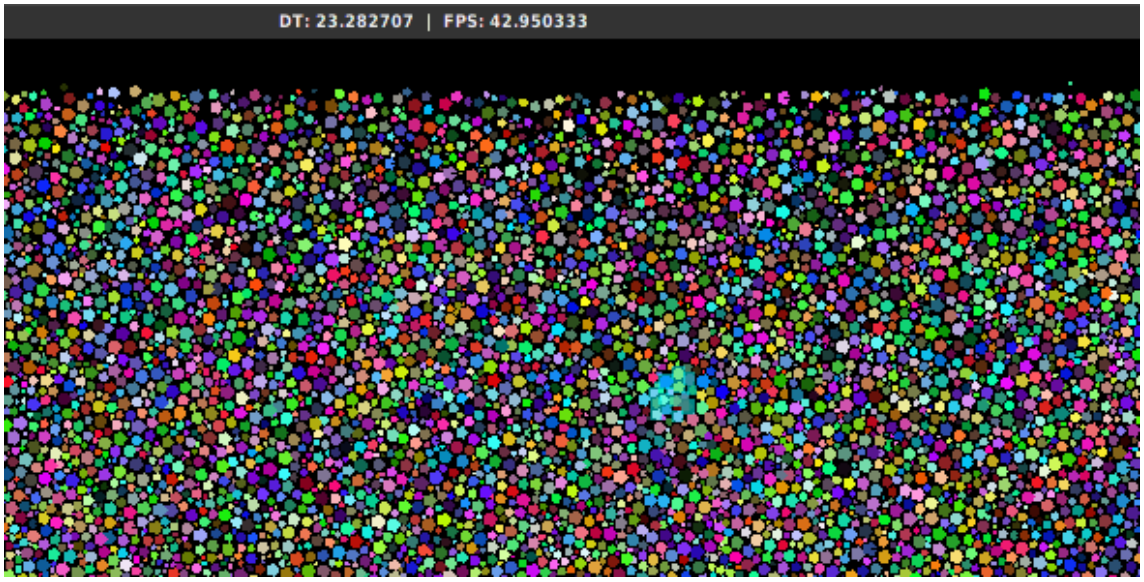


Figure 16.3: Particle simulation: screenshot - spatial partitioning cell

16.4.2 Benchmarks

The computer used to benchmark the particle simulation has the following hardware specifications:

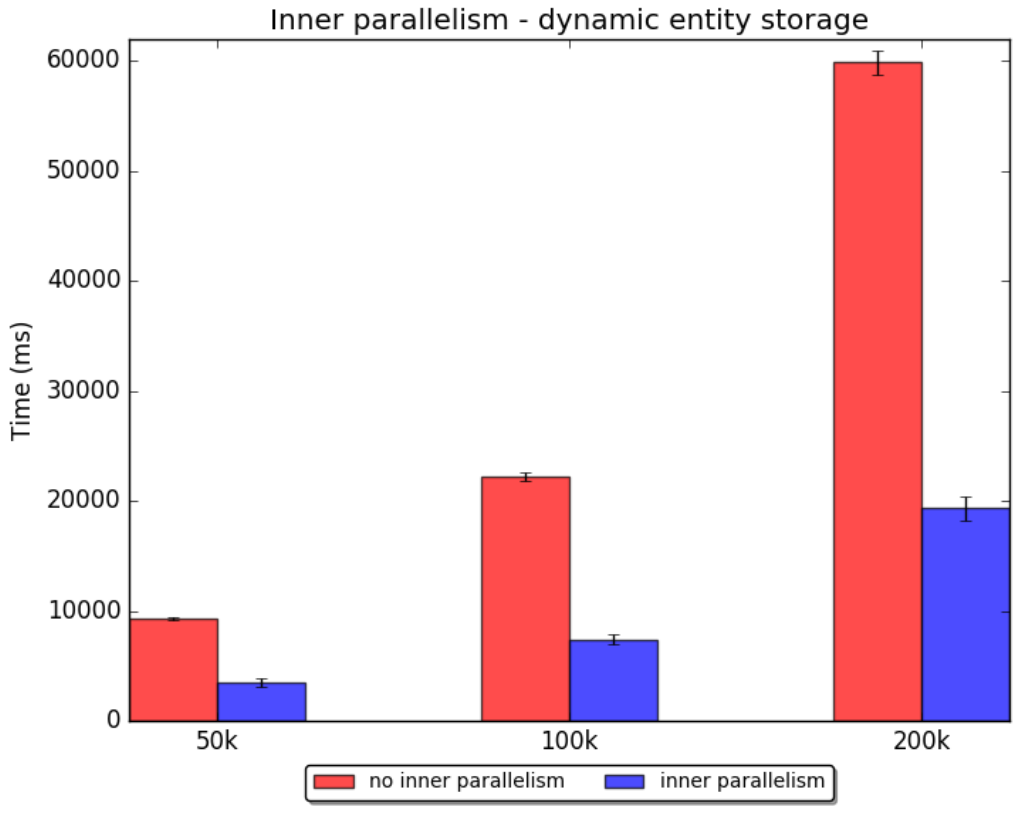
- CPU: **Intel® Core™ i7-2700K Processor** *(8M Cache, up to 3.90 GHz)*;
- RAM: **HyperX Beast 16GB DDR3-2400MHz**;
- Motherboard: **ASRock Z77 Extreme4-M**.

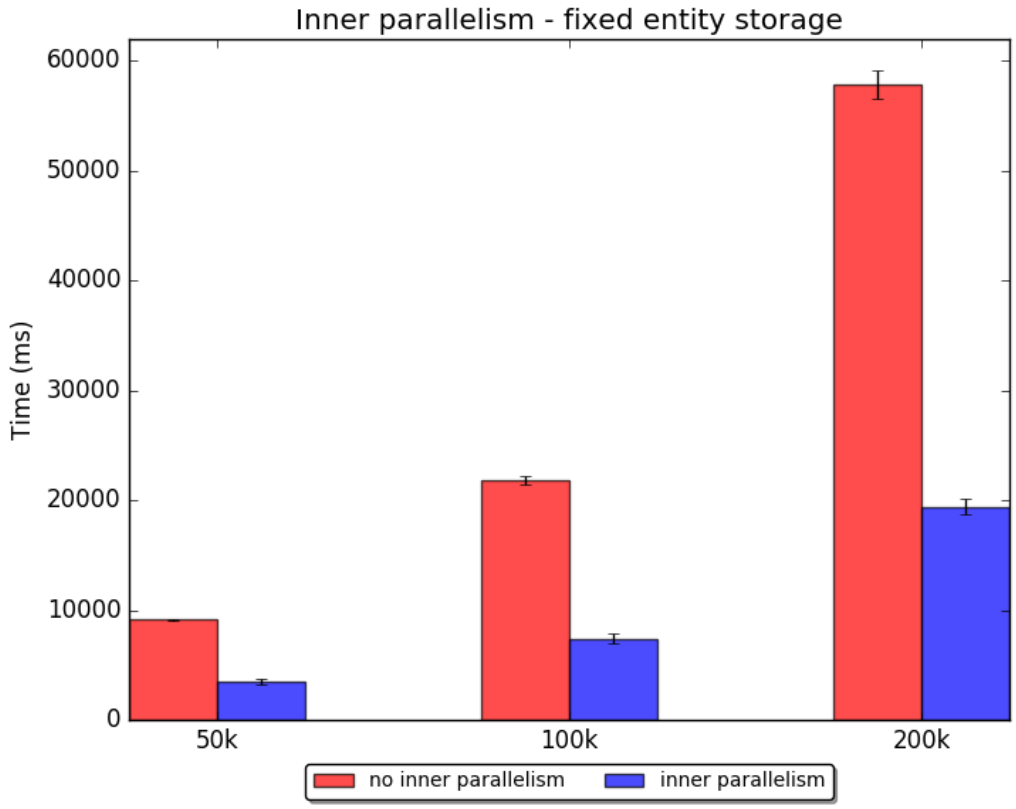
The simulation was executed on a system with **Arch Linux x64** as its operating system, using 8 worker threads. Rendering was disabled for the benchmarks.

`g++` version 6.1.1 was used, with the following compiler flags: `-Ofast -march=native -ffast-math -ftree-vectorize`.

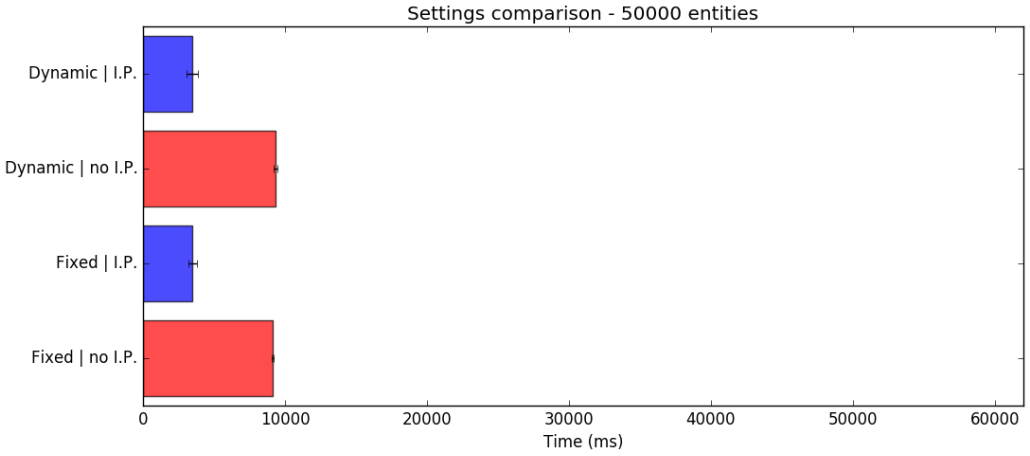
The error bars in the following graphs represent the *standard deviation*.

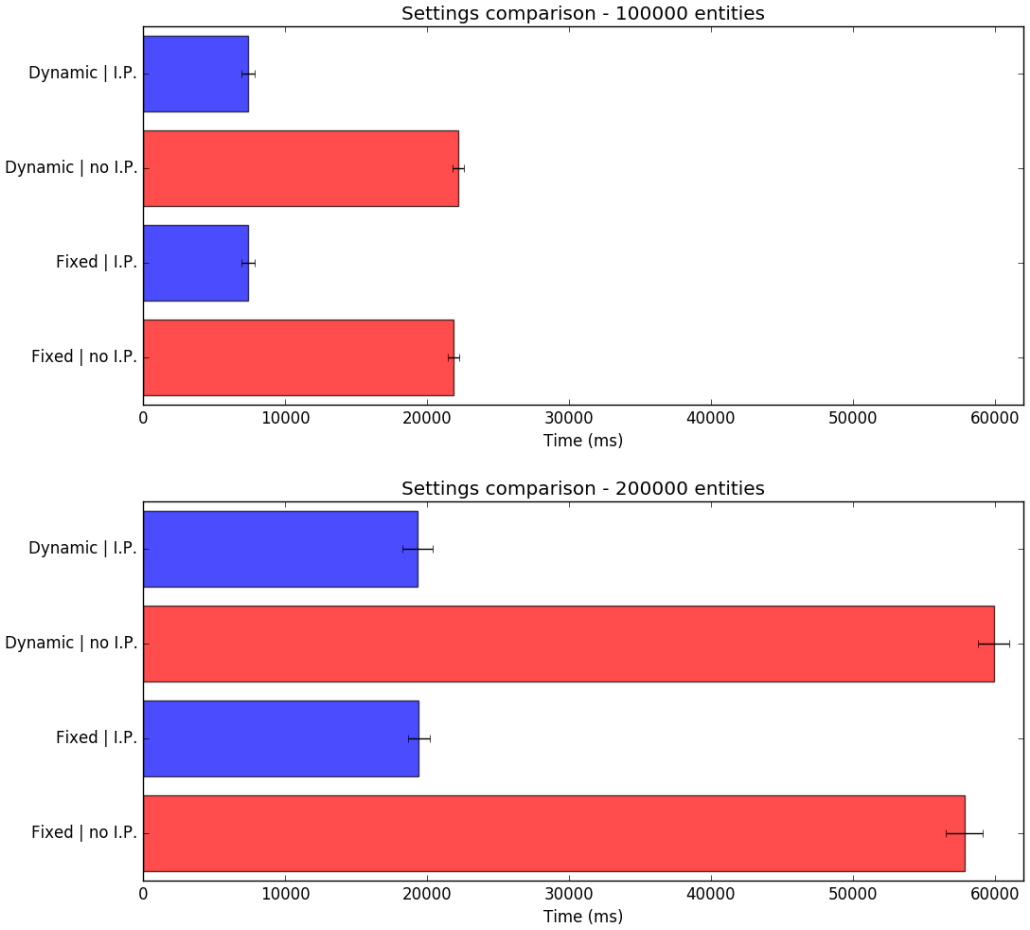
16.4.2.1 Dynamic versus fixed entity storage





16.4.2.2 Entity scaling





16.5 Conclusions

The following conclusions can be deduced from the benchmark graphs:

- **Fixed** entity storage seems slightly faster than **dynamic** entity storage when inner parallelism is disabled, possibly due to the fact that checks for possible reallocations during entity creations are not present. However, dynamic entity storage seems slightly faster than fixed entity storage when inner parallelism is enabled, even if entity creation only occurs sequentially at the beginning of the simulation. The results on **fixed versus dynamic** entity storage are therefore inconclusive with this benchmark - a different simulation, where entities are continuously created and destroyed over time, may compare the two entity storage strategies more fairly. The tables below shows the execution time percent change of the dynamic storage strategy:

	Fixed (no I.P.)	Dynamic (no I.P.)
50k	baseline	+2.28%
100k	baseline	+1.42%
200k	baseline	+3.56%

	Fixed (I.P.)	Dynamic (I.P.)
50k	baseline	-0.45%
100k	baseline	-0.26%
200k	baseline	-0.44%

- As expected, splitting system execution in multiple subtasks using **inner parallelism** results in a huge run-time performance boost. On the machine used for the benchmarks, an average 65% relative performance increment is achieved:

	No inner parallelism	Inner parallelism
50k	baseline	-62.32%
100k	baseline	-66.38%
200k	baseline	-67.07%

The complete source code of the example particle simulation can be found in the following GitHub repository, under the *Academic Free License ("AFL") v. 3.0*: https://github.com/SuperV1234/bcs_thesis.

Chapter 17

Entity creation/destruction benchmark

17.1 Description

The benchmark aims to measure the performance of continuous real-time entity creation/destruction with various combinations of compile-time settings. At the beginning of the application, a fixed number of entities is created. The entities possess a `c::life` component that will destroy them after a random amount of time, creating a new entity instance on destruction. The entity replication process is limited by a counter stored in `c::life` that is decreased on creation.

As with the previous example application, multiple simulations are executed and benchmarked, combining the following compile-time options and parameters:

- Entity count: 50000, 100000 and 200000;
- Inner parallelism: **enabled** or **disabled**;
- Entity storage strategy: **fixed** or **dynamic**.

In total, 12 simulations are executed.

17.2 Components

The only existing component type is `c::life`.

- **Life**: controls the lifetime of an entity and its amount of replications.

```
1    struct life
2    {
3        float _v;
4        int _spawns;
5    };
```

17.3 Systems

- **Life**: deals with entity lifetime and replication. Continuously decreases every particle's `life` value and marks particles as dead when their lifetime is over. Once an entity is marked as dead, a **deferred function** that creates a new particle is enqueued if `c::life::_spawns` is greater than 0.
  - Multithreading is enabled.

17.3.1 System implementations

The commented implementation of the `s::life` system is provided below:

```
1  struct life
2  {
3      template <typename TData>
4      void process(ft dt, TData& data)
5      {
6          data.for_entities([&](auto eid)
7              {
8                  // Alias the entity's lifetime value.
9                  auto& l = data.get(ct::life, eid)._v;
10
11                 // Alias the entity's left replications value.
12                 auto& spawns = data.get(ct::life, eid)._spawns;
13
14                 // Decrease the entity's lifetime.
15                 l -= 10.f * dt;
16
17                 // If the lifetime value reaches zero...
18                 if(l <= 0.f)
19                 {
20                     // ...mark the entity as dead.
21                     data.kill_entity(eid);
22
23                     // If the entity can replicate itself...
```

```
                    if(spawns > 0)
                    {
                        // ...enqueue a deferred function creating
                        // a new entity with one less replication.
                        data.defer([spawns](auto& proxy)
                            {
                                mk_particle(proxy, spawns - 1);
                            });
                    }
                }
            });
    }
};
```

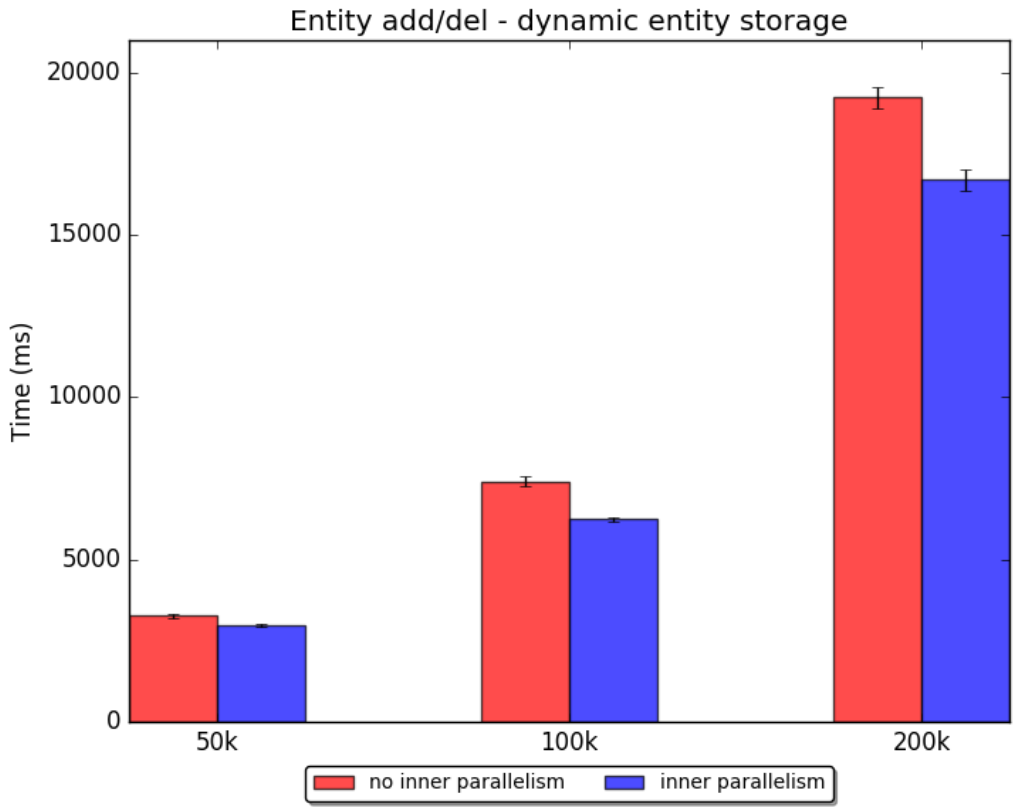
17.4 Results

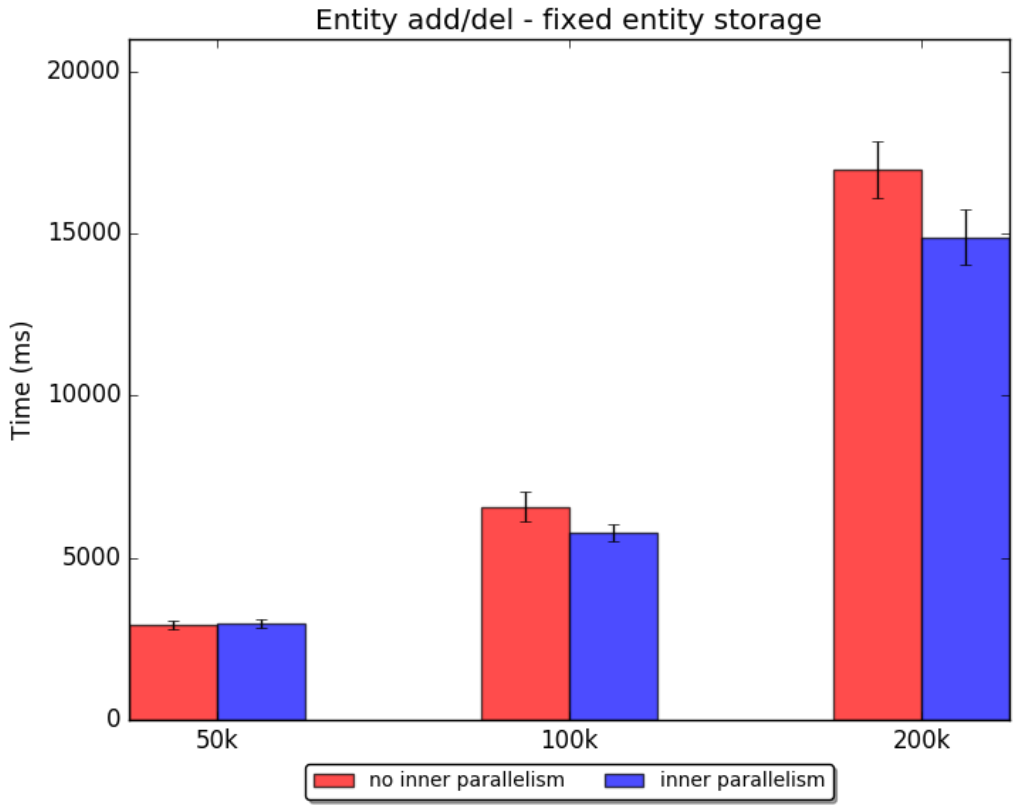
17.4.1 Benchmarks

The previously described machine and environment were used for this simulation.

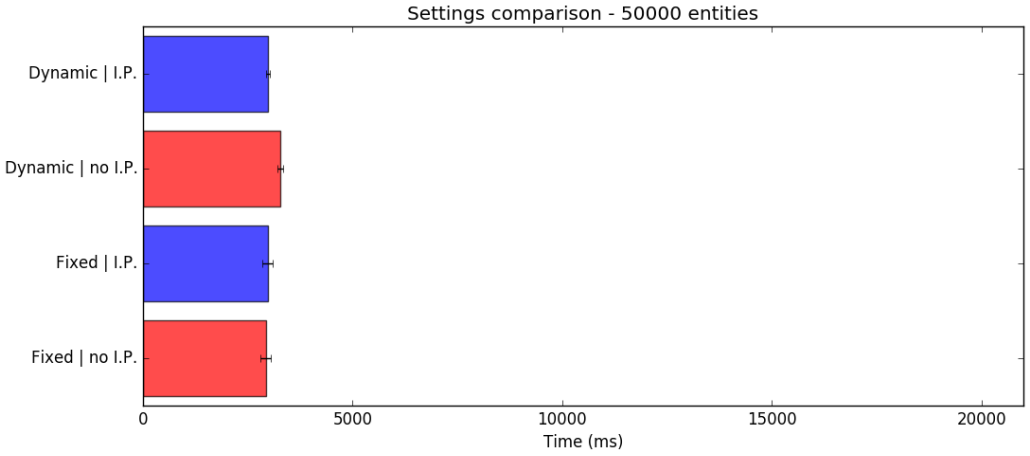
Again, the error bars in the following graphs represent the *standard deviation*.

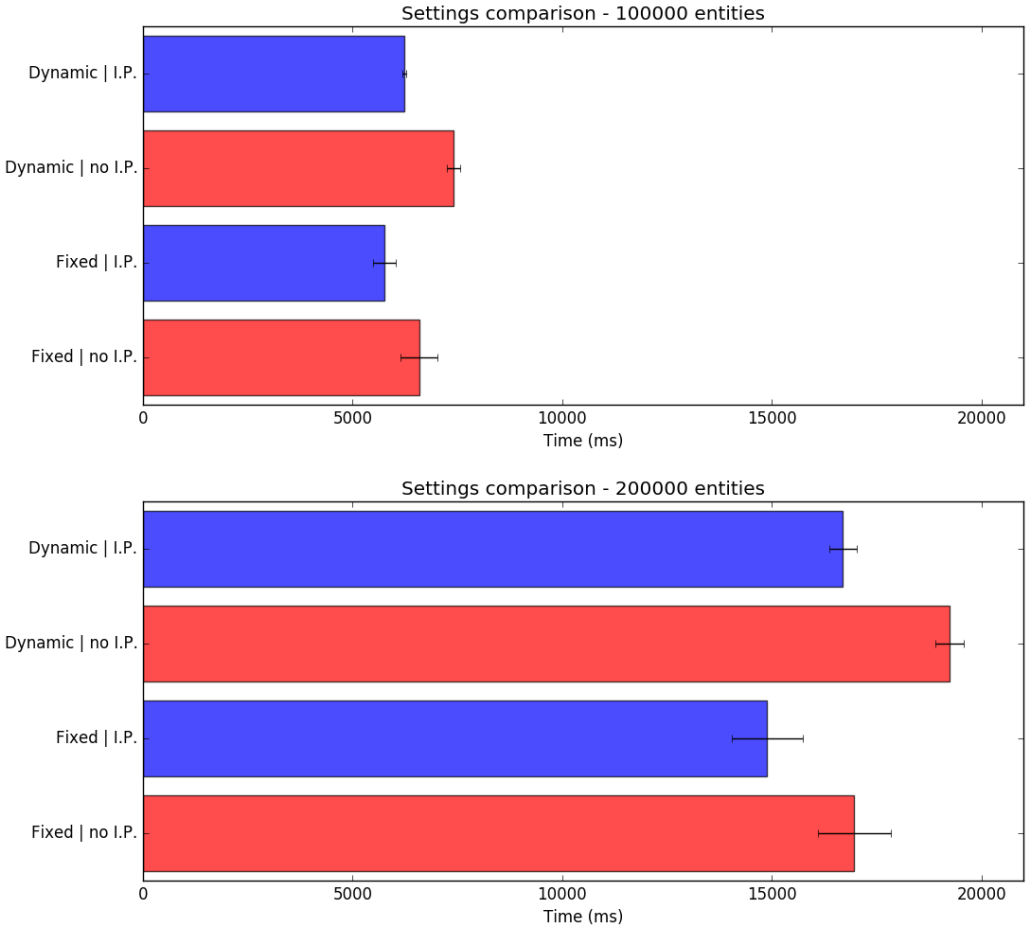
17.4.1.1 Dynamic versus fixed entity storage





17.4.1.2 Entity scaling





17.5 Conclusions

The following conclusions can be deduced from the benchmark graphs:

- **Fixed** entity storage is definitely slightly faster than **dynamic** entity storage, indepedently of inner parallelism settings. The presence of reallocation checks *(in dynamic entity storage)* produces a noticeable run-time overhead due to the huge number of entity creations/destructions. The tables below shows the execution time percent change of the dynamic storage strategy:

	Fixed (no I.P.)	Dynamic (no I.P.)
50k	baseline	+11.77%
100k	baseline	+12.56%
200k	baseline	+13.34%

	Fixed (I.P.)	Dynamic (I.P.)
50k	baseline	+0.47%
100k	baseline	+8.19%
200k	baseline	+12.09%

- Again, using **inner parallelism** results in an expected run-time performance boost. Due to the nature of this simulation, however, only an average 10% relative performance increment is achieved compared to the previous simulation's 65% relative performance increment:

	No inner parallelism	Inner parallelism
50k	baseline	-3.98%
100k	baseline	-14.23%
200k	baseline	-12.75%

The complete source code of the example entity creation/destruction benchmark can be found in the following GitHub repository, under the *Academic Free License ("AFL") v. 3.0*: https://github.com/SuperV1234/bcs_thesis.

List of figures

References

[1] J. Gregory, *Game engine architecture, second edition.* CRC Press, 2014.

[2] A. Kirmse, *Game programming gems 4*, vol. 4. Charles River Media, 2004.

[3] K. Pallister, *Game programming gems 5*, vol. 5. Charles River Media, 2005.

[4] M. Dickheiser, *Game programming gems 6*, vol. 6. Charles River Media, 2006.

[5] M. Doherty, "A software architecture for games."

[6] D. Wiebusch and M. E. Latoschik, "Enhanced decoupling of components in intelligent real-time interactive systems using ontologies," in *Software engineering and architectures for realtime interactive systems (searis), proceedings of the ieee virtual reality 2012 workshop*, 2012.

[7] T. Dahl, T. Koskela, S. Hickey, and J. Vatjus-Anttila, "A virtual world web client utilizing an entity-component model," in *2013 seventh international conference on next generation mobile apps, services and technologies*, 2013, pp. 7–12.

[8] R. Nystrom, "Game Programming Patterns - Decoupling Patterns - Component," 2014. [Online]. Available: http://gameprogrammingpatterns.com/component.html.

[9] A. Martin, "T-machine.org category archives - Entity Systems." [Online]. Available: http://t-machine.org/index.php/category/entity-systems/.

[10] A. Martin, "Entity Systems are the future of MMOG development – Part 2," 2007. [Online]. Available: http://t-machine.org/index.php/2007/11/11/entity-systems-are-the-future-of-mmog-development-part-2/.

[11] T. Leonard, "Postmortem: Thief: The Dark Project," 1999. [Online]. Available: http://www.gamasutra.com/view/feature/3355/postmortem_thief_the_dark_project.php.

[12] S. Bilas, "A Data-Driven Game Object System," 2002. [Online]. Available: http://scottbilas.com/games/dungeon-siege/.

[13] M. West, "Evolve Your Hierarchy," 2007. [Online]. Available: http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/.

[14] T. Cohen, "A Dynamic Component Architecture for High Performance Gameplay," 2010. [Online]. Available: http://www.insomniacgames.com/a-dynamic-component-architecture-for-hi

gh-performance-gameplay/.

[15] Ike, "Is it reasonable to build applications (not games) using a component-entity-system architecture? - Answer," 2016. [Online]. Available: http://programmers.stackexchange.com/a/306983/78524.

[16] B. Bucklew, "IRDC US 2015 - Data-Driven Engines of Qud and Sproggiwood," 2015. [Online]. Available: https://www.youtube.com/watch?v=U03XXzcThGU.

[17] cppreference, "Derived classes - Virtual base classes." [Online]. Available: http://en.cppreference.com/w/cpp/language/derived_class#Virtual_base_classes.

[18] E. Truyen, W. Joosen, B. N. Jørgensen, and P. Verbaeten, "A generalization and solution to the common ancestor dilemma problem in delegation-based object systems," in *Dynamic aspects workshop (daw04)*, 2004, vol. 6.

[19] "No Bugs" Hare, "C++ for Games: Performance. Allocations and Data Locality," 2016. [Online]. Available: http://ithare.com/c-for-games-performance-allocations-and-data-locality/.

[20] T. Albrecht, "Pitfalls of Object Oriented Programming," 2009. [Online]. Available: http://harmful.cat-v.org/software/OO_programming/_pdf/Pitfalls_of_Object_Oriented_Programming_GCAP_09.pdf.

[21] Improbable, "Improbable - SpatialOS - Learn more," 2016. [Online]. Available: https://improbable.io/learn-more.

[22] L. Dionne, "Boost.Hana documentation." [Online]. Available: http://www.boost.org/doc/libs/1_61_0/libs/hana/doc/html/index.html.

[23] P. F. II, "Dependent typing in C++," 2015. [Online]. Available: http://pfultz2.com/blog/2015/01/24/dependent-typing/.

[24] V. Romeo, "'std::bitset' inclusion test methods," 2015. [Online]. Available: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0125r0.html.

[25] A. Martin, "Data Structures for Entity Systems: Contiguous memory," 2014. [Online]. Available: http://t-machine.org/index.php/2014/03/08/data-structures-for-entity-systems-contiguous-memory/.

[26] cppreference, "Empty base optimization." [Online]. Available: http://en.cppreference.com/w/cpp/language/ebo.

[27] A. Martin, "Entity ID's: how big, using UUIDs or not, why, etc?" 2015. [Online]. Available: http://t-machine.org/index.php/2015/06/09/entity-ids-how-big-using-uuids-or-not-why-etc/.

[28] cppreference, "'std::enable_if'." [Online]. Available: http://en.cppreference.com/w/cpp/types/enable_if.

[29] cppreference, "Overload resolution." [Online]. Available: http://en.cppreference.com/w/cpp/language/overload_resolution.

[30] V. le Clément, P. Schaus, C. Solnon, and C. Lecoutre, *Sparse-sets for domain implementation.*

2013.

[31] P. Praxis, “Sparse Sets,” 2012. [Online]. Available: https://programmingpraxis.com/2012/03/09/sparse-sets/.

[32] M. Semenov, “Fast Implementations of Sparse Sets in C++,” 2015. [Online]. Available: http://www.codeproject.com/Articles/859324/Fast-Implementations-of-Sparse-Sets-in-Cplusplus.

[33] V. A. Burns, “GitHub: ‘Diana‘,” 2013. [Online]. Available: https://github.com/discoloda/Diana.

[34] V. Romeo, “GitHub: ‘vrm_core‘.” [Online]. Available: https://github.com/SuperV1234/vrm_core.

[35] B. Stroustrup, G. D. Reis, and A. Sutton, “‘Static If ’ Considered,” 2013. [Online]. Available: http://open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3613.pdf.

[36] V. Voutilainen, “Static if resurrected,” 2015. [Online]. Available: http://open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4461.html.

[37] V. Voutilainen, “constexpr_if,” 2015. [Online]. Available: http://open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0128r0.html.

[38] J. Maurer, “constexpr if: A slightly different syntax,” 2016. [Online]. Available: http://open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0292r0.html.

[39] cppreference, “explicit (full) template specialization.” [Online]. Available: http://en.cppreference.com/w/cpp/language/template_specialization.

[40] B. Wicht, “Simulate static_if with C++11/C++14,” 2015. [Online]. Available: http://baptiste-wicht.com/posts/2015/07/simulate-static_if-with-c11c14.html.

[41] G. Diago, “Proposal for techniques that improve ad-hoc generic code: static_if emulation,” 2015. [Online]. Available: https://github.com/isocpp/CppCoreGuidelines/issues/353.

[42] V. Romeo, “C++Now 2016: ”Implementing ‘static‘ control flow in C++14”,” 2016. [Online]. Available: https://github.com/SuperV1234/cppnow2016/tree/master/static_control_flow.