P. J. Safarik University

Faculty of Science

A CONCURRENT COMPONENT-BASED ENTITY ARCHITECTURE FOR GAME DEVELOPMENT

Field of Study:	**Informatika**
Institute:	**Ústav informatiky**
Tutor:	**RNDr. Jozef Jirásek, PhD**

Košice 2015 **Ferdinand Majerech**

Thanks

I'd like to thank my supervisor, RNDr. Jozef Jirásek, PhD, for advice that helped keep this work from going away on random tangents. I'd also like to thank developers of open source projects which allowed me to avoid reinventing the wheel, especially Sam Lantinga (SDL), Michael Parker (Derelict) and Walter Bright and Andrei Alexandrescu (D programming language).

Abstract

With the prevalence of large simulated game worlds in game development, component-based approaches to game entities are becoming more common. At the same time, gaming hardware is becoming increasingly parallel, making it more difficult to fully utilize the hardware, especially when writing games for multiple hardware platforms. We describe a new Entity-component system design, focusing on scalable, automatic parallelization with little or no need for manual management of threads.

Keywords: game development, entity, component, ECS, entity-component system, parallelization, multithreading, scheduling

Table of contents

Introduction

With advancement of gaming hardware, mainstream computer games are increasingly becoming "simulated worlds" rather than traditional abstract "games". These worlds contain a large number of objects of many different kinds; and these objects interact with each other and change in real time.

Traditional programming paradigms are often unable to elegantly and maintainably represent the variety of objects and usually do not fully exploit modern CPU architectures and memory hierarchies. A few decades ago, it was possible to create a "fast enough" game that would get faster new CPUs came to market. Today this is no longer the case speed of individual CPU cores is improving slowly and manufacturers have turned to increasing core counts as a way to improve performance.

Today's games need to adapt to the fact that different users may have varying numbers of CPU cores of varying power. With varying core counts it is no longer trivial to fully utilize the power of a CPU, especially in real-time applications such as games where commonly used concurrent programming techniques may introduce unacceptable latency.

Since late 2000s there has been a increase in use of entity-component systems (`ECS`) and data-driven design in the gaming industry, replacing OOP entity architectures. These approaches allow to represent a wide variety of dynamically changing game objects in a maintainable way. Some ECS designs also lead to efficient implementations taking advantage of modern hardware, e.g. with sequential storage of aligned fixed-size data items, making it possible to write game code that is predictable by modern CPUs and avoids cache misses.

Entity-component systems come with their own issues. Efficient ECS implementations written in C or C++ usually work outside of the language type system, requiring users to write bug-prone code working with untyped data and to rely on coding conventions instead of the guarantees of a static type system. Processing of same data in multiple passes during a single game update often leads to unexpected dependencies between passes, making refactoring difficult. Most importantly, it's non-trivial to write multithreaded game code in an entity system, especially when core counts of target machines vary.

We propose a modified ECS design that takes advantage of multi-core CPUs by automatically distributing game load between CPU cores. We also provide a reference implementation of this design as a library written in the D programming language and measure the performance and scalability of this implementation with a game-like benchmark.

Goals

Our main goal is to design an ECS framework capable of scaling to large CPU core counts (ideally on the order of tens of cores) by running ECS Systems (Processes) in separate threads (depending on core count), without requiring the programmer to manage threads manually. To enable such scaling we propose modifications to the ECS concept, most importantly the separation of component state into read-only **past** and mutable **future** state, as described in **Chapter 2**. We use this modified ECS as a base for a concurrent ECS design, detailed in **Chapter 3**.

Another goal is to provide an implementation of the concurrent ECS (as an open-source ECS framework), which is described in **Chapter 4**. We then use this implementation to create a simple game to test the implementation and benchmark its scalability with with different CPU core counts (**Chapter 5**).

1 Background

This chapter serves as an introduction to game entities and entity systems and describes terminology used in this work. This is not a complete reference; we focus on topics needed to understand our work. Other topics are only briefly mentioned.

1.1 Game entity

A game entity (*Entity*) is an object existing in a game world or simulation. It should not be confused with an OOP object; though a game entity can be *implemented* as an OOP class instance. Examples of entities include: player avatar, tree, enemy, weapon, particle system; but also abstract objects such as a sound effect source, a "factory" entity that spawns other entities[1], an invisible trigger object that starts a scripted sequence when touched, and so on. Depending on type of a game, the number of entities may reach hundreds (adventures, simple FPS), thousands and tens of thousands ("big" RTS, open world RPG, space sims), or even essentially unbounded (MMOs). Game entities and their evolution are described in more detail in [4].

An entity consists of state that belongs to the entity (position, health...) and may use assets (resources) shared by multiple entities; e.g. all enemies with the same 3D model or sound effects, all actors with the same AI script, and so on.

OOP class entities

The most obvious way to implement entities in OOP languages is through a class hierarchy. In such a hierarchy, every 'kind' of entity is implemented as a separate type derived from a parent Entity class. OOP entities often lead to deep, hard to refactor hierarchies and make it difficult to fully exploit the processing power of modern hardware. Today, class entities are becoming less common, mostly due to increasing popularity of various component-based approaches. The case against OOP-style class entities is explained further in [14, 1].

[1] Not to be confused with the Factory design pattern.

Component-based entities

In games using the component approach, data and functionality is separated into *components* aggregated to form an entity. There is no entity class hierarchy; an entity is a container of components. Different entities may consist of different components as illustrated in Figure 1.1; in OOP terms, every entity 'instance' behaves as an instance of a different 'class'. Components may be implemented e.g. as OOP classes, plain-old-data types (e.g. C structs) or even groups of related variables. Various component approaches are described in [13, 6, 7].

Figure 1.1: Entities from game *Crysis* [18] described in terms of components

Entity as an ID

Taking this trend to conclusion, there is no need for an 'Entity' structure at all; an entity can be reduced to a unique identifier. Components can be stored in a data structure that retrieves components when given an entity ID.

1.2 Entity-component systems

In Entity-Component Systems or `ECS`[2], entities are simple IDs that may not even be stored in memory. ECS components are data structures associated with entity IDs, with little or no logic; game logic is separated into `Systems`, which can be thought of as "logic components".

In an ECS, individual components do not need to refer to, depend on or even know about each other; there is no separate dependency resolution step after the components are initialized. If some combination of different components is needed to achieve a goal, functionality implementing that goal is isolated into a System which "selects" the components together (see below). Separation of code and data in an ECS also enables efficient implementations minimizing memory latency through Data-oriented design. The concept of an ECS is described in more detail in [12, 11].

Component

A component in an ECS is a 'dumb' data structure. It may be implemented as a plain old data type in C/C++/D, a Java bean, etc. An entity may contain 0 or 1 components of any given type. Most components are small data structures storing data for a single concept or feature; e.g. transformation of an object in 3D space or health/damage information.

[2] Also called Entity Systems, Entity Frameworks and so on; there is no standard terminology as of this writing. To make things worse, unrelated component-based approaches are sometimes referred by the same name.

Example entities and their components

Entity ID	Components
#0	$Physics_0$, $Visual_0$
#1	$Visual_1$, $Sound_1$
#2	$Sound_2$
#3	$Physics_3$, $Visual_3$, $Sound_3$

Table 1.1: 4 example entities with components of types *Physics*, *Visual* and *Sound*. We will refer to these entities in following sections.

System

A System is an independent logic block that processes components of specific types, implementing a part of a game update such as collision detection, movement or rendering. It may be implemented e.g. as a class with a member function to process the components, and may contain data other than components (e.g. as class data members). During a game update, the ECS finds entities that contain **all** components required by the system and passes these components to the System. Most Systems should be simple pieces of code that "do one thing and do it well". In the ideal (not necessarily common) case a System should always run the same sequence of instructions with little or no branching.

Example of a System running in an ECS

RenderSystem: requires Physics and Visual components

Entity ID	Processed	Components
#0	**Yes**	$\mathbf{Physics}_0$, $\mathbf{Visual}_0$
#1	No	$\mathbf{Visual}_1$, $Sound_1$
#2	No	$Sound_2$
#3	**Yes**	$\mathbf{Physics}_3$, $\mathbf{Visual}_3$, $Sound_3$

During a game update, the ECS determines that entities #0 and #3 contain both Physics and Visual components. For each of these entities, it executes the RenderSystem, passing the Physics and Visual components.

Term: Game update

This chapter often uses the term *game update* or *frame* - a single update of all state in the game simulation. A game update usually takes a short amount of time (e.g. 33.3 ms at 30 FPS or 16.7 ms at 60 FPS). During an update, state of all entities will be updated based on passed time (e.g. entities can be moved, collisions detected, events can be triggered, etc.). Usually, the game is also drawn to the screen (which is why the word *frame* is also used).

RDBMS analogy

ECS are sometimes also called RDBMS-style entity systems as an ECS is similar in concept to a relational database table.

An *entity* acts as a **row** of a database table where *component types* are **columns**. If an entity contains a component of a specific type, the column for that type in the row of the entity contains data. If the entity does not contain such component, the cell is `NULL`. A *System* behaves like a **SELECT** selecting specific component types in all entities that contain components of these types.

EntityID	Physics	Visual	Sound
#1	$Physics_0$	$Visual_0$	NULL
#2	$Physics_1$	NULL	$Sound_1$
#3	NULL	NULL	$Sound_2$
#4	$Physics_3$	$Visual_3$	$Sound_3$

Table 1.2: Entities from Table 1.1 as a database table

Data-oriented design and ECS

Data-oriented design (`DOD`) is a way of thinking about high-performance program design that aims to bridge the expanding gap between CPU speeds and memory latency by centering program design around memory layout to fully utilize modern CPU features such as cache hierarchies. DOD is explained in more detail in [1, 10].

An ECS is easy to implement in data-oriented style, and performance of such implementations has been a major argument in support of ECS. Some examples of using DOD in an ECS design would be: storing components of one type in a large contiguous

array; aligning addresses and sizes of individual components to cache line size, and so on. A System, when executed, can then sequentially iterate such arrays without jumps in memory, avoiding cache misses. If a System is simple and contains little or no branching, it is easier to predict for the CPU, further minimizing stalls.

ECS issues

Despite the advantages, entity-component systems come with their own problems.

An obvious way to parallelize an ECS is to divide Systems into groups that modify disjunct sets of component types, and running each group in a separate thread. However, it is often difficult to create equally sized groups; often there are a few groups affecting most component types that will fully use a small number of CPU cores while smaller groups finish early and leave most cores unused. Also, the total number of groups may be small, limiting the number of useful cores. Our intention is to modify the ECS concept to enable better scaling to increasing core counts.

If a System modifies component data read by another System in the same update, accidental dependencies/gameplay effects may occur. Changing the order of Systems may then break the game. This problem is also common with non-component entity approaches. To run Systems in parallel we need to avoid or solve this problem, as we cannot rely on a particular run order.

Existing ECS frameworks

This is a short (and incomplete) list of existing open source ECS frameworks.

Framework	Language (ports)	License	Notes
Artemis [15]	Java (C++, C#, D ...)	New BSD	Popular framework
Ash [16]	AS3 (JS, Ruby, Java ...)	MIT	Popular in dynamic languages
EntityX [19]	C++	MIT	Template based design
Entreri [20]	Java	New BSD	High performance

Table 1.3: Open source ECS frameworks

2 Modified ECS design

In this chapter we describe a modified *non-concurrent ECS*, which serves as a base for our concurrent ECS design. This ECS design uses specific memory organization and disallows certain usage patterns to avoid most causes of data races that can occur if Systems run in separate threads. This makes it possible to parallelize the ECS with little synchronization (locking) overhead.

We separate component state into immutable *past* and mutable *future* state to allow each System to run in a separate thread without locking components processed as the System is executed. We also use modified terminology to better reflect the fact that the Systems may run in separate threads:

- **Process**: We use the term *Process* to refer to a System in our ECS design to make it more evident that we refer to a logic block that may run in its own thread or CPU core. This term is always capitalized (Process) to disambiguate from e.g. OS processes. The Ash ECS framework [16] also uses this term.
- **Past state**: *Past* state is read-only game state (components) generated by the previous game update. Processes (Systems) can *only* read past state. As it's not writable, there is no need to lock past state when Processes read it.
- **Future state**: *Future* state is game state being generated by Processes during the current game update.

Concepts

This chapter describes groups of types in terms of generic programming *Concepts* [17], implementable through C++ concepts or D template constraints. Concepts describe requirements on a type, such as a Process or component type; these requirements may be more complex than what is allowed by OOP *interfaces* (e.g. a concept may require a signature of a type member function to satisfy a predicate). In some languages, *type classes* fulfill a similar role.

2.1 Sources of data races

In an ECS, a game update consists of executing all Systems. A System is executed by finding all entities containing all components the System requires and for every

matching entity, passing those components to the System, which may modify them.

The order the Systems are executed in must be known and must not change; if Systems A and B both modify the same component, the game may behave differently depending on whether A runs before or after B.

Now consider what would happen if we naively ran A and B in parallel.

Each Process would need to lock each entity or component to avoid cases where A and B would simultaneously touch the same entity, which could cause massive overhead with thousands of entities and tens of Processes. For example, with 2000 entities, 50 processes and an average of 3 components accessed by a Process, we would need at least 300000 locks per frame; which is 900000 locks per second at 30 FPS.

Even if we ensured that A and B never modify the same component, we would hit a problem with Process execution order. If A and B run in parallel, we cannot guarantee that they always process a component in the same order (at least not without further synchronization overhead). Game logic may behave differently for individual entities.

We can distill the information in preceding paragraphs into two main issues making efficient parallelization of an ECS at Process level difficult:

- **Synchronization overhead**: If an entity (components) can be modified by one Process and read or modified by another, it must be locked. Locking is expensive.
- **Nondeterminism**: A game update starting with certain game state may end in different state depending on the order Processes process individual entities.

Past state

In our previous experience [21] with an ECS implementation, we noticed that a Process usually modifies very few of the components it processes; many Processes do not even modify any components (those that generate output). With a naive approach we would need to lock these components or the entire entity.

We can avoid locking these unmodified components by keeping a read-only copy of components from the previous game update. Processes can then use read-only versions of most components without locking; only components that may be modified need

to be locked. This approach could reduce collisions between Processes compared to locking whole entities or individual components. However, by itself this would not remove the overhead of locking each entity for components that *are* modified, nor would it make the ECS process entities in a deterministic manner.

Past and future

The idea of immutable state from the previous game update leads to a concept of separate *past* and *future* state. Instead of processing some components as read-only and some as read-write, we can explicitly separate components processed by a Process into read-only inputs (**past** components) and write-only outputs (**future** components). This avoids nondeterminism; regardless of their order, Processes read the same input data and can therefore produce the same outputs.

Synchronization is still needed with future state, as multiple Processes may modify the same future component. There is also a new problem; if different Processes produce different future versions of the same component, which is the "correct" version?

We chose a trivial, but effective solution; while a Process may any past components, we do not allow two Processes to write the same future component. There is always only one future version of any component, and there is no need to lock future components (assuming the implementation keeps them separate). With this we can avoid per-entity synchronization. Implementation needs to enforce this limitation, either at compile-time or runtime. This does come make it less trivial to use of the ECS; our test game should provide a reference on how to write game code with this limitation in mind.

2.2 Memory organization

This section describes at high level how game state is stored as seen by internals of the ECS. Implementations may use further optimizations (e.g. paged buffers, cache line aligned components) that may result in actual memory layout being different.

If there are n component types, we have buffers $components_i$ and $componentCounts_i$ for $i \in (0, \ldots, n-1)$. $components_i$ stores all components of i-th component type. If the number of entities is m, and $j \in (0, \ldots, m-1)$, $componentCounts_i[j]$ specifies the

number of components of type i in entity j. This value must be in 1 or 0, though some implementations may use greater values to allow multiple components of a type.

Components in $components_i$ are stored in the order of entities, without gaps. If an entity does not contain a component of type i, there is no element corresponding to that entity in $components_i$. The index in $components_i$ of a component of type i belonging to entity j is $\sum_{k=0}^{j-1} componentCounts_i[k]$; that is, a sum of component counts of that component type for all preceding entities.

For example, the 4 entities shown in Table 1.1 wound be stored as follows, assuming component types are numbered as $Physics = 0, Visual = 1, Sound = 2$:

$components_0 = (Physics_0, Physics_1, Physics_3)$, $componentCounts_0 = (1, 1, 0, 1)$
$components_1 = (Visual_0, Visual_3)$, $componentCounts_1 = (1, 0, 0, 1)$
$components_2 = (Sound_1, Sound_2, Sound_3)$, $componentCounts_2 = (0, 1, 1, 1)$

All these buffers are duplicated for past and future state, i.e. there are separate $components_pi, componentCounts_pi$ for past and $components_fi, componentCounts_fi$ for future components. Implementations can usually reuse memory by swapping past and future buffers between updates.

2.3 Component concept

For completeness, this section describes the concept of a component in our ECS (it is similar to component in other ECS's, as described in Chapter 2). Reflection-based requirements are described in comments, as implementations will vary based on programming language.

Concept 1 *Component*

```
// - Must be copyable by a byte-by-byte copy (no elaborate copyconstructor).
// - Must have no destructor.
// - Default values data members must be known at compile-time,
//   and a default-initialized component must be valid.
// - Must not have any indirections (e.g. pointers). (some implementations may want to
//   relax this limitation; the intention is to prevent a component "owning" external data)
// - Must have one or more per-instance data members

// Required: A compile-time integer that uniquely identifies the component type
// (implementations may use a different type identification mechanism).

enum uint ComponentTypeID
```

Example component type satisfying this concept:

```
/// Accelerates and decelerates entities.
struct EngineComponent:
    // Unique ID of this component type
    enum ushort ComponentTypeID = 1
    /// Acceleration of the engine.
    float acceleration = 0.0f
    /// Max speed the entity can be accelerated to by this engine (in any direction).
    float maxSpeed = 0.0f
```

2.4 Process execution

Process in detail

Processes are executed by the ECS, not directly by the user. Each Process is an aggregate with a `process` member function, which, during a game update, is called for each entity that has all components required by the Process.

Processes can be implemented with non-virtual member functions if the ECS is aware of Process types at compile-time, e.g. by a type parameter. This even allows to generate custom code to execute each Process. In languages where this is not possible, Processes can be implemented less efficiently e.g. as an inheritance hierarchy.

Concept 2 *Process*

```
// Past components passed as const (read-only) references.
// Future component passed as a writable reference.
void process(/*optional*/ ref Context, ref const Past1,
             ref const Past2, ..., ref const PastN, /*optional*/ref Future)
```

The `process()` method takes $N >= 1$ *past* component and 0 or 1 *future* component parameters. Past parameters specify the types required by the Process, i.e. `process()` will only be called on entities that contain components of types $(Past1, Past2, ..., PastN)$. One Process can only write future components of one type, or none at all (see *Past and future* in Section 2.1). state as we make the ECS concurrent. The optional `Context` parameter can provide access to data such as the entity ID and implementation-specific operations. Past components are passed as `const`; the `process()` function cannot modify them. They are also passed by reference, which is not strictly necessary; however, most implementations will this to avoid copying overhead. It *is* necessary to pass the future component by reference to allow the Process to specify future state.

Process execution loop

At high level, each Process is executed in a loop that iterates over buffers storing past component types required by the Process and future component type written by the Process. This is illustrated in Figure 2.2. One iteration of the loop corresponds to one entity in the ECS; all entities are iterated.

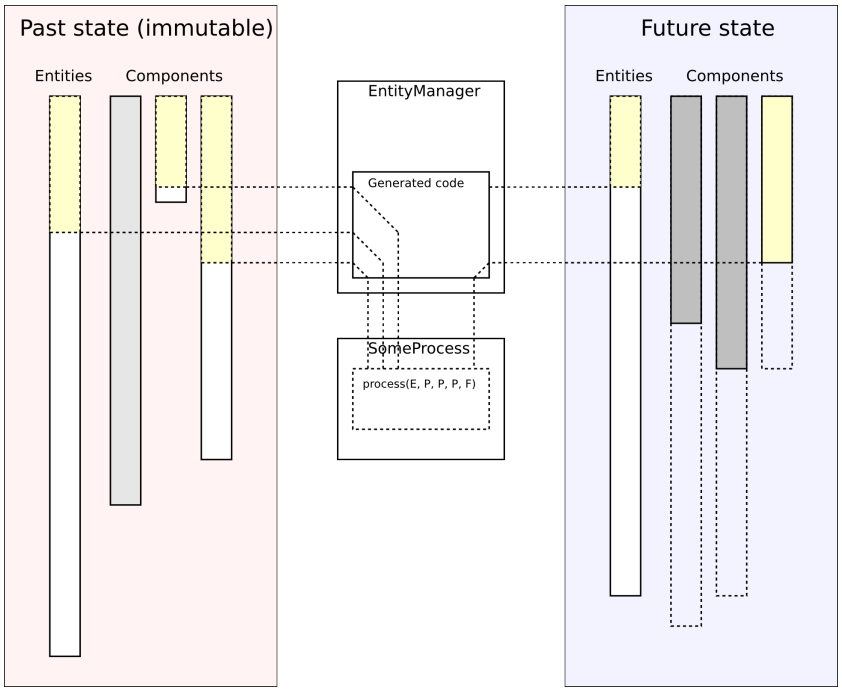

Figure 2.1: The process execution loop sequentially iterates over one or more past component buffers and at most one future component buffer. When all components required by the Process are present in an entity, they are passed to the Process.

During an iteration, the current entity is checked for past components required by the Process; if they are all present, the past components and an address to write the future component to are passed to the Process.

Algorithm 1 *executeProcess(process)*

```
// Globals or class members existing outside of the algorithm
Component0[] componentsP0, Component1[] componentsP1, ..., ComponentN-1 componentsPN-1
Component0[] componentsF0, Component1[] componentsF1, ..., ComponentN-1 componentsFN-1
bool[ComponentTypeCount] componentCountsP
bool[ComponentTypeCount] componentCountsF
uint entityCount

---

// (Compile-time) tuple of past component types required by process
alias PastT = pastComponentTypes(process)
// Note that the number of past component types can be known at compile-time
alias n = PastT.length
// Type of the future component written by process
alias FutureT = futureComponentType(process)

// pastIndices is PastT.length long; pastIndices[i] is the index of the component
// (if any) of i-th type in the current entity.
uint[] pastIndices = [0, ..., 0]
// Index of the future component in the current entity
uint futureIndex = 0

for entityIdx in 0 .. entityCount:
   // Update component indices to point to components in the current entity
   // by adding component counts.
   updatePastComponentIndices(entityIdx, pastIndices, componentCountsP)

   pastIndices[0] += componentCountsP[PastT[0].index][entityIdx]
   ...
   pastIndices[n - 1] += componentCountsP[PastT[n - 1].index][entityIdx]

   // Check if the current entity has all past components required by process
   if componentCountsP[PastT[0].index] && ... && componentCountsP[PastT[n - 1].index]:
      // Reference to the future component for the process to write to
      ref FutureT future = componentsF${FutureT.index}[futureIndex]
      // Execute process on the current entity
      // Pass references to past components from component buffers corresponding to
      // component types read by the Process.
      // If the Process requires a Context parameter, that would be passed here as well.
      process.process(passPast(pastIndices, componentsP1 ... componentsPN-1), future)

      componentCountsF[FutureT.index][futureIndex] = 1

      ++futureIndex
```

The above algorithm is a generalization; for example, a version of the algorithm executing a Process that does not write any future components will not keep track of a future component index, and a version executing a Process where the `process` function has a `Context` parameter will pass that parameter as well. As we process individual entities, we keep track of indices of past components and the future component, if any. These indices point to $components_i$ buffers for i corresponding to the past component types required by the Process. `passPast` is not necessarily a function; it signifies that the implementation should pass the past components as referenced by `pastComponentIndices`.

While some time is wasted by touching every single entity, processing data sequentially allows implementations to take full advantage of modern CPU caches, branch prediction and microoptimizations of various CPU architectures. Overall, the overhead of the process execution loop is $O(n)*perEntityOverhead+O(m)*(passComponentsOverhead+usefulTime)$, where n is the number of entities, m is the number of entities with matching the Process (having all required components), *perEntityOverhead* is the overhead needed to process one entity that may or may not match the Process (this includes the matching itself), *passComponentsOverhead* is the extra overhead for matching entities (time spent passing components to the Process), and *usefulTime* is the actual time spent by the `process` function of the Process.

2.5 The game update

Some maintenance work must run in every game update besides executing Processes.

1. Past and future state are swapped to reuse resources.
2. "Dead" entity IDs are removed, and their components ignored so they are not regenerated during the next update.
3. Newly created entities are added to the ECS (with their components) as past state.
4. Once the ECS is parallelized, process scheduling must run (see Chapter 3)

3 Concurrency in the entity system

In previous chapter, we have described an ECS implementation designed to be easy to parallelize. In this chapter we how actually make the ECS parallel. Processes in our ECS are executed in a tight loop (described in section 2.4) to make best use of hardware such as CPU caches. We treat the Process execution loop as the base execution unit; distributing Processes among multiple threads during each game update, but not distributing the execution of a single Process.

3.1 Process threads

At the beginning of a game update, we assign each Process to one of a fixed number of threads; this number does not change at runtime. During the update, these threads run in parallel, executing Processes. A Process may only be moved from one thread to another *between* updates.

Besides scaling with CPU core count we need game updates to run very fast without spikes (lag); 33.3 ms (30 FPS) or 16.7 ms (60 FPS) is often targeted. To avoid thread creation/destruction overhead, we only create threads at startup. This results in a need for some synchronization of these threads.

Process thread execution

Each process thread runs an infinite event loop that moves between four states; possible states are shown in Figure 3.1 and explained below. Process thread execution is illustrated in Figure 3.2 .

`Waiting`: The thread does not write any data, allowing the main thread to run code needed between game updates (e.g. scheduling). Ideally, `Waiting` should be a loop frequently checking for a change to `Executing` to avoid latency. In practice, this may not lead to best results (see section 4.3, class `EntityManager.ProcessThread`).

`Executing`: The thread runs Process execution loops for Processes assigned to it. The Process execution loop reads past state and writes to at most one future component buffers per Process, allowing the thread to avoid synchronization in this state.

`Stopping`: The thread is being stopped (joined back to the main thread).

`Stopped`: The thread is not running (has been joined).

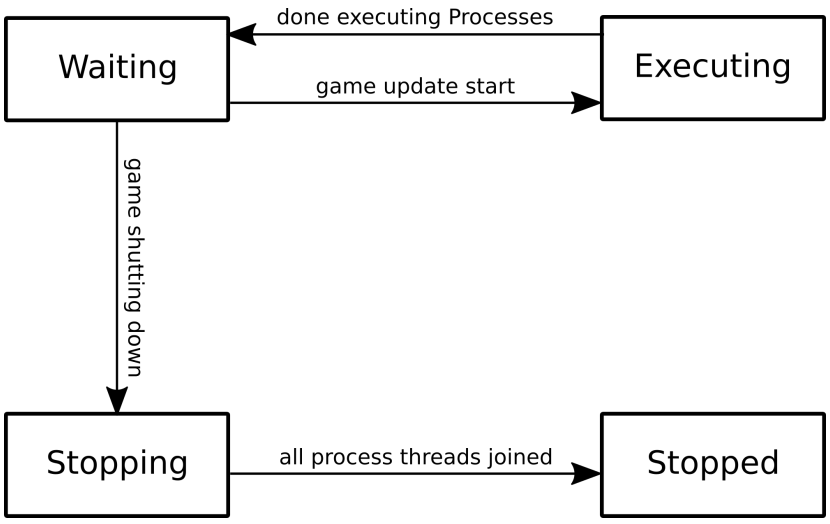

Figure 3.1: Process thread states.

The **main thread** executes Processes, but it also has a managing role; at the beginning of an update, the main thread assigns Processes to threads (see also section 3.2) and changes state of each process thread to `Executing`. During update, the main thread runs Processes assigned to it. When finished, it waits until all process threads are `Waiting`. When the game is shutting down, the main thread changes state process threads to `Stopping` and joins them.

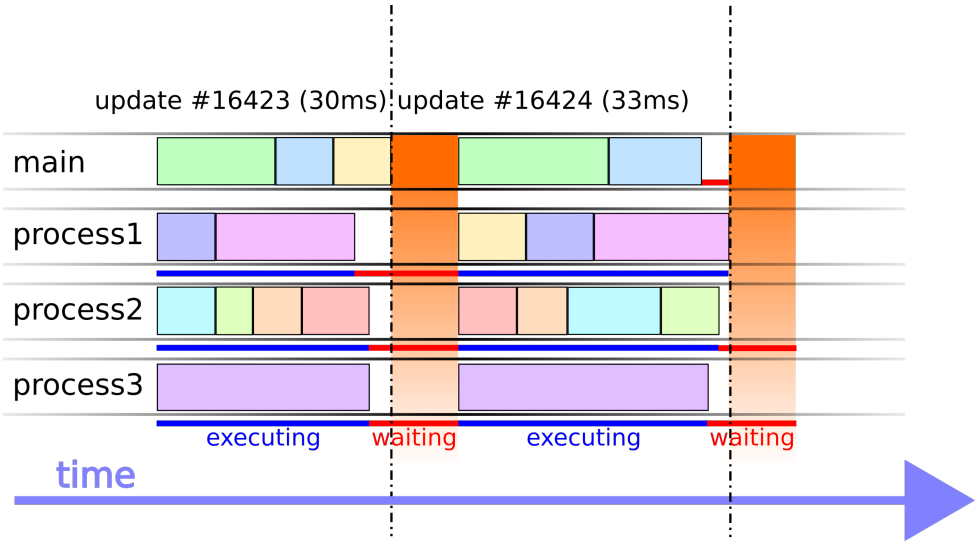

Figure 3.2: Example of threaded execution (main thread and 3 process threads) during two game updates. The orange gradient shows the non-parallel overhead (such as scheduling) of the main thread between updates.

The only points where threads need to be synchronized are when starting execution at the beginning of an update and when waiting for all threads to finish executing at the end. This can be implemented without much overhead e.g. by using atomic operations to access a variable representing the state.

3.2 Process scheduling

To gain improved performance from parallelizing Process execution, each thread must spend roughly the same time to run its Processes during an update. If we randomly assign Processes to threads we may end up with a situation where one thread is executing and all the other threads are waiting after their Processes finished early.

If we measure time spent executing each Process during a game update, we can use that information to assign Processes to threads so that the time spent by each thread is similar. At first, we are going to assume that each Process will take the same time to run in the next game update as it took in the previous update, which does not always reflect reality; section 3.3 adresses this problem.

We need to assign Processes to threads very quickly; We run scheduling between game updates, which is the part of ECS execution that is not parallel. If we consider 1 ms to be acceptable time spent in non-parallel code (recall that a game update may take e.g. 16 or 33 ms), 0.1 ms can be acceptable for scheduling. This means we need a fast algorithm; the results can be suboptimal as long as they are "good enough".

Multiprocessor scheduling

Multiprocessor scheduling [22] is an NP-hard problem, commonly applied in operating systems for scheduling tasks. This is a simple version of the multiprocessor scheduling problem for our case:

We have n jobs with execution times $(p_1, p_2, \ldots p_n)$, and m identical machines. If J_j is the set of execution times of jobs scheduled to machine j, $t_j = \sum p \in J_j$ is the *load* of machine j. $l = max(\{t_j, j \in (1, 2, \ldots, m)\})$ is then the total execution time of a schedule. The goal of the multiprocessor scheduling problem is to schedule jobs (Processes) to machines (threads, cores) in such a way that the total execution time

of the schedule is minimal.

Algorithms applicable to multiprocessor scheduling

The multiprocessor scheduling problem is closely related to problems such as job-shop scheduling, number partition and bin-packing. Researching algorithms applicable to these problems can help find approaches applicable to our problem.

We use the fast greedy longest processing time (LPT) algorithm and the slightly more complicated COMBINE [9] algorithm, which produces a better approximation. As explained in section 3.3, the suboptimality of result is overshadowed by the lack of predictability of Process execution time.

LPT algorithm

Note

In pseudocode of this and the following algorithms, `processes.length` refers to n and `threadCount` refers to m from the Multiprocessor scheduling problem description.

Algorithm 2 *LPT(processes, threadCount)*

```
// struct Process:
//     int id
//     int executionTime
//     int assignedThread
// Process[] processes;

// Sort processes in decreasing order (O(processes.length * log(processes.length)))
processes.sort!((a, b) => a.executionTime > b.executionTime);
// Fill a priority queue (lower time - higher priority) with thread IDs with cost 0
PriorityQueue!((a, b) => a < b, int, int) threads;
for t in 0 .. threadCount:
    threads.push(t, 0)

// Assign Processes to threads in order of descending execution time (O(processes.length))
for p in 0 .. processes.length:
    // Assign to the thread with least cost so far (O(1))
    (int thread, int cost) = threads.front
    processes[p].assignedThread = thread
    // O(log(threadCount))
    threads.changePriority(thread, cost, cost + processes[p].executionTime)


return processes
```

LPT is straightforward to implement, and will always produce a result within $\frac{4}{3} - \frac{1}{3m}$

[8] of optimum with a time complexity of $O(n \cdot log(n))$ assuming $n >= m$ (greater or equal number of Processes compared to cores. If there are more cores than Processes, we can trivially assign every Process to a separate core). This should scale well even for very complex games with hundreds of Processes.

COMBINE (and MULTIFIT)

Algorithm 3 *MULTIFIT(processes, threadCount, upperTimeBound, lowerTimeBound)*

```
// preconditions:
// * must be able to create a schedule with execution time < upperTimeBound
// * processes is sorted in decreasing order

// assignedThreads[i] is the index of the thread Process i is assigned to
int[processes.length] assignedThreads
// MAX_ITERATION is a fixed constant (7 in our implementation)
for iteration in 0 .. MAX_ITERATION:
    int[threadCount] threadUsage // total time of Processes in each thread
    threadUsage[] = 0
    int capacity = (upperTimeBound + lowerTimeBound) / 2
    bool failedToFit = false

    // FFD(First Fit Decreasing) with capacity determined above
    // From longest to shortest process
    nextProcess: for p in 0 to processes.length
        for t in 0 to threadCount:
            // Do we fit into this thread?
            if threadUsage[t] + processes[p].executionTime <= capacity:
                threadUsage[t] += processes[p].executionTime
                assignedThreads[p] = t
                // We did fit this Process into this thread within capacity,
                // go fit the next Process
                continue nextProcess // continue the outer for loop
        // We failed to fit this Process to any thread within capacity.
        failedToFit = true
        break;

    if failedToFit:
        lowerTimeBound = capacity
    // if failedToFit is false, we've assigned threads for all Processes
    else:
        for p in 0 .. processes.length:
            processes[p].assignedThread = assignedThreads[p]
        upperTimeBound = capacity
// Note: if neither iteration can fit the processes into capacity,
// we don't affect assigned threads of Processes.

return processes
```

MULTIFIT works as a binary search over bin size (`upperTimeBound`. `lowerTimeBound`, `capacity`), where each iteration attemts to pack Process times using bin-packing with the first-fit decreasing algorithm. MULTIFIT will always find a schedule within

$1.22 + (\frac{1}{2})^k$ [5] of optimum where k is `MAX_ITERATION` in above pseudocode, which is an improvement compared to LPT. However, in empirical tests [9], LPT often generates better schedules than MULTIFIT. This disadvantage is avoided by combining MULTIFIT with LPT in the COMBINE algorithm.

Algorithm 4 *COMBINE(processes, threadCount)*

```
processes = LPT(processes, threadCount)
int lptMakespan = max thread usage of any thread (e.g. access priority queue from LPT)
// Total process execution time divided by thread count
int avgThreadTime =
    ceil(processes.map!(p => p.executionTime).sum / cast(double)threadCount)
// if the LPT result is greater than this, it is optimal (see [10])
if lptMakespan >= 1.5 * averageExecutionTime:
    return processes

// processes is sorted in the order of decreasing execution time
const slowestProcessTime = processes[0].executionTime
const lptErrorBound = 4.0 / 3 - 1 / (3.0 * threadCount_);

// MULTIFIT tries to find a better total time (makespan) than LPT by using the LPT
// result as the upper time bound. The lower time bound is a maximum of theoretical
// best cases.
processes = MULTIFIT(processes, threadCount, lptMakespan,

                    max(lptMakespan / lptErrorBound, slowestProcessTime, avgThreadTime));
```

The COMBINE [9] algorithm combines the worst-case approximation bound of MULTIFIT with an output that is never worse compared to LPT. It works by generating a schedule with LPT and then using the makespan of that schedule as an upper time bound for a MULTIFIT run, which will either generate a better schedule than the LPT run, or none at all. Time complexity of COMBINE is $O(n \cdot log(n) + kn \cdot log(m))$ [9] where k is `MAX_ITERATION` in MULTIFIT, which is $O(n \cdot log(n))$ assuming $n >= m$.

3.3 Process execution time estimation

In section 3.2, we assumed that we know exact Process execution times. This is not always the case; execution time of a Process depends on many factors, e.g. the number of matching entities, how long Process logic takes depending on component values,

player input, even external factors like the OS scheduler, dynamic CPU clocking and so on. We are not able to precisely predict Process execution time in such a complex environment. We can, however, approximate it to some degree to (mostly) avoid some imprecisions that would lead to worst-case scheduler outputs.

Consider the behavior of (any) scheduling algorithm if we underestimate the execution time of a Process. The scheduling algorithm will try to schedule Processes so that the load of each thread is balanced, which may e.g. lead to one thread running a few "long" Processes while another executes a larger number of "shorter" Processes. But if one of these "shorter" Processes takes more time than estimated (and especially if this happens with *more* than one Process in a thread), one thread will spend more time executing, prolonging the entire game update. This may happen irregularly between game updates, leading to short updates randomly interspersed with unexpectedly long updates - microlags, which will be seen by the player as jitter.

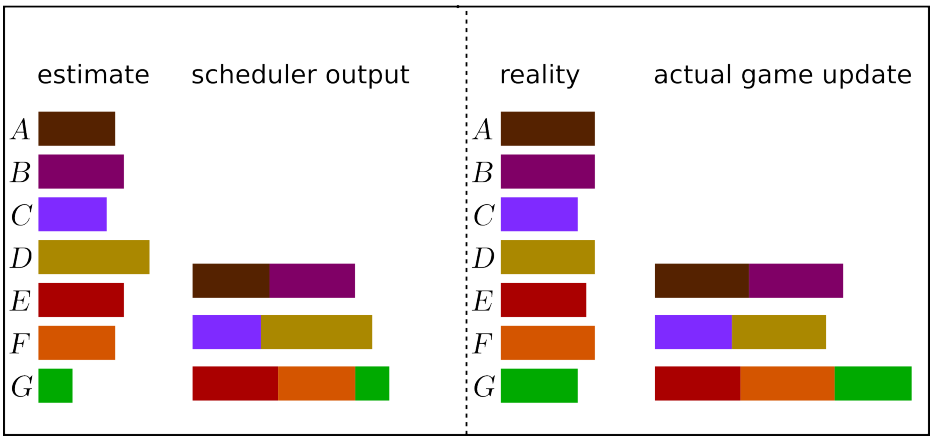

Figure 3.3: Underestimating Process execution time may cause occasional longer game updates, leading to irregular microlag (jitter). In this example, a major underestimate of process *G* results in thread *3* taking longer to execute, prolonging the update.

In the opposite scenario, execution time is overestimated. Processes expected to take "long" will be scheduled to run in less loaded threads. The main negative effect of this is that more "short" Processes are moved to other threads. If some of these are *under*estimated, that may lead to jitter, but if most estimates are *over*estimates with very few underestimates, most threads will take less time than expected from the schedule. This will only lead to a shorter (jitter) game update if *all* threads take less time than expected; as long as the overestimates are not very common and large, framerate should be smooth. If a minimum update time is enforced (e.g. by waiting), update time will not even be affected as long as that minimum is not exceeded.

An example where overestimates are useful is a Process that has runs mostly smoothly with occasional spikes when it takes much longer. If the estimator expects the Process to always take close to the 'smooth' time, whenever the Process hits a spike there will be a long game update, potentially resulting in jitter. However, if the time estimator assumes it will take the close to the 'spike' time, the scheduler will do its best to minimize update time by moving the Process into a less loaded thread, and distributing the workload from that thread among (possibly multiple) other threads. This will decrease lag when the Process does spike, while it will increase average update time to a lesser degree. Overall, the result is smoother framerate.

Time estimation problem

We have n Processes. During each game update, we measure execution times of these Processes, and run the time estimator algorithm with these measured times as input. The output is an array of execution time estimates for all Processes.

Various approaches may need to track other data, e.g. data about best/worst/average time for each Process, machine learning data, etc. For example, our (simple) time estimator algorithm uses time estimates saved from the previous game update.

Time estimation algorithm

Algorithm 6 *ExecutionTimeEstimate(processes, previousEstimates, estimateFalloff)*

```
// on the first game update, previousEstimates contents are zeroed
// estimateFalloff is a real number between 0 and 1

int[processes.length] timeEstimates
timeEstimates[] = 0

for p in 0 .. processes.length:
    // If execution time measured from the previous game update was longer than
    // estimated, estimate for the next update will be increased to the measured value.
    if processes[p].executionTime >= previousEstimates[p]
        timeEstimates[p] = processes[p].executionTime
    // If execution time is shorter than estimated, the next estimate will be shortened
    // by (difference from estimate) * estimateFalloff.
    else:
        int diff = previousEstimates[p] - processes[p].executionTime
        timeEstimates[p] = previousEstimates[p] - cast(int)(diff * estimateFalloff)

return timeEstimates
```

The execution time estimation algorighm used is very simple; for each Process, compare its execution time measured (m) in the previous game update with its estimated time for that update (e). If $m >= e$ (execution time was underestimated), estimate that the Process will take m time in the next update. This means that when during a spike, the estimate quickly increases to avoid underestimating for the following game updates. If $m < e$ (overestimate), the next estimate will be decreased by $(e-m) \cdot \mathit{estimateFalloff}$. This means that after an overestimate, the estimate gradually decreases proportionally to the overestimate error. Thus after a spike the estimate will stay "high" for a few updates to avoid underestimate on any following spikes, but will eventually return to a "normal" value. $\mathit{estimateFalloff} \in [0, 1]$ is a user-specified constant, in our case we use *0.2* but better values may exist, and what value produces the best results may even depend on the details of each game.

4 Implementation

We use the D programming language[3] for implementation due to its high performance, generic programming through templates and metaprogramming features. The implementation should be translatable to C++ with some effort, while languages using generics such as C# and Java, or languages with no generic programming features such as C would likely require a major redesign to avoid degrading performance.

Full source code is available on the attached DVD, and also online[4]. All source code is released under the Boost Software License 1.0[5].

The ECS itself is split into two libraries: `tharsis-core` and `tharsis-full`. The former contains core ECS implementation (entity/Process management, past and future state, threading, scheduling/time estimation, type information/constraints, etc.) while the latter wraps `tharsis-core` and adds various utility functionality needed to use the framework, such as component/Process types useful for most games, entity serialization code and so on. Our work focuses on `tharsis-core`.

4.1 Most important subsystems

D syntax

Following code snippets are written in the D programming language. Some snippets are simplified to help readability/reduce amount of text.

D features used that are uncommon in other programming languages:

- Compile-time (template) parameters: Type or value parameters passed at compile-time that affect code generation. Syntax: `Type!Param`, `Type!(Param1, Param2)`, `function!(CompileTimeParam)(RunTimeParam)`
- `scope(exit){}` blocks: code in the block is ran when current scope is exited, even in case of failure (exception being thrown)
- `static if{}`: Compile-time `if`; code in the `if` block is only generated if the condition is true.
- `a => b`: lambda function syntax

[3] http://dlang.org

[4] https://github.com/kiith-sa?tab=repositories

[5] http://www.boost.org/users/license.html

Entity manager

class EntityManager(Policy) defined in module `tharsis.entity.entitymanager` is the "core" subsystem of Tharsis. It registers Processes, runs process threads and contains the entry point to game update execution as well as most of the management between game updates.

The most important functions of `EntityManager` are:

- `public void startThreads()`: Called at the beginning of the game to create and start threads - `ProcessThread`. May throw an exception in case of an error.
- `public void executeFrame() nothrow`: Executes the full game update, including entity management, scheduling and any other code that must run before or after the game update.

  Function:

```
void executeFrame() @trusted nothrow
{
    // Get the past & future component/entity buffers for the new frame.
    GameState!(Policy)* newFuture = cast(GameState!(Policy))past_;
    GameState!(Policy)* newPast   = future_;

    // Copy alive past entities to future.
    newPast.copyLiveEntitiesToFuture(*newFuture);

    // Allocate space for the newly added entities.
    const addedEntityCount = (cast(EntitiesToAdd)entitiesToAdd_).prototypes.length;
    newPast.addNewEntitiesNoInit(addedEntityCount);
    newFuture.addNewEntitiesNoInit(addedEntityCount);
    // Preallocate future component buffers if needed.
    newFuture.preallocateComponents(componentTypeMgr_);

    // Add the new entities into the reserved entity/component space.
    initNewEntities(entitiesToAdd_.prototypes, componentTypeMgr_, *newPast,
                    *newFuture);
    (cast(EntitiesToAdd)entitiesToAdd_).prototypes.clear();

    // Save changes back to data members.
    future_ = newFuture;
    past_   = cast(immutable(GameState!(Policy)*))(newPast);

    scheduler_.updateSchedule!Policy(processes_, diagnostics_);
    // Process execution in threads happens here.
    executeProcesses();
    updateDiagnostics();

}
```

Important `EntityManager` data members used in the snippet above:

- `immutable(GameState!(Policy))* past_` and `GameState!(Policy)* future_`: past and future game state; see the *Game State* subsection below.
- `shared(EntitiesToAdd) entitiesToAdd_`: entities to add at the beginning of a game update. `shared` means this synchronized and cast to be used.
- `AbstractComponentTypeManager componentTypeMgr_`: see the *Type Management* section below.
- `Scheduler scheduler_`: see the *Scheduling* section below.

- `public void registerProcess(P)(P process) nothrow`: Registers a Process of type `P`. Generates code to run the Process (`runProcess`); that code instantiates `EntityRange!(EntityManager, P)` and generates more code. Wraps it all in a `ProcessWrapper`.
- `void runProcess(P)(EntityManager self, P process) nothrow`: Function generated by `registerProcess()` that implements the process execution loop (see section 2.4).

  Function:

```
void runProcess(P)(EntityManager self, P process) nothrow
{
    // Iterate over all living entities, executing the process on those that
    // match the 'process' methods of the Process.
    for(auto entityRange = EntityRange!(typeof(this), P)(self);
        !entityRange.empty(); entityRange.popFront())
    {
        // If P writes a future component, we need to specify that there are no
        // future components of this type for this entity even if we don't match it.
        static if(hasFutureComponent!P)
        {
            entityRange.setFutureComponentCount(0);
        }

        // If the entity has all past components read by P, pass components to
        // P.process().
        if(entityRange.matchComponents!(pastComponentIDs!(P.process)))
        {
            callProcessMethod!(P.process)(process, entityRange);
        }
    }

}
```

- `void executeProcesses() @system nothrow`: Execution in the main thread. Also see `void run()` in `ProcessThread` below, which details execution of the Process threads.

  Function:

```
void executeProcesses() @system nothrow
{
    import core.atomic;
    // Ensure any memory ops finish before finishing the new game update.
    atomicFence();

    // Start running Processes in ProcessThreads (state: Waiting -> Executing)
    foreach(thread; procThreads_)
    {
        thread.startUpdate();
    }
    // Execute Processes assigned to the main thread (index 0).
    executeProcessesOneThread(0);

    // Wait till all threads finish executing (atomic reads of 'state').
    while(procThreads_.canFind!(e => e.state == ProcessThread.State.Executing))
    {
        // Give the OS some time. Explained in ProcessThread.run description below
        yieldToOS();
    }

}
```

The **Policy** parameter of `EntityManager` wraps compile-time parameters for code generation in `EntityManager` and its subsystems; see Policy-based design [2].

class EntityManager.ProcessThread represents a thread handling Process execution. The most important `ProcessThread` members are:

- `shared(State) state_`: Current state of the thread, as described in section 3.1 . This is a 1-byte value accessed only with atomic instructions.
- `EntityManager self_`: A reference to the `EntityManager` - which contains wrapped Processes.
- `@property State state() nothrow`: Reads `state_` atomically.

- `void run() nothrow`: Thread execution, alternating between waiting and executing Processes assigned to the thread.

  Function:

  ```
  void run() nothrow
  {
     for(;;) final switch(this.state)
     {
         case State.Waiting:
             // Wait for the next game update. We need to give the OS some time.
             while(this.state == State.Waiting)
             {
                 yieldToOS();
             }
             break;
         case State.Executing:
             // scope(exit) ensures the state is set even if we're crashing
             scope(exit) { atomicStore(state_, State.Waiting); }
             // Execute Processes assigned to this thread.
             self_.executeProcessesOneThread(threadIdx_);
             // Ensure any memory ops finish before finishing a game update.
             atomicFence();
             break;
         case State.Stopping:
             //  finish the thread.
             return;
         case State.Stopped:
             assert(false, "run() still running when the thread is Stopped");
     }

  }
  ```

  We let the thread sleep while in the `Waiting` state. At first we used hot-loop waiting, which resulted in higher performance but frequent microlag as the OS unpredictably paused our threads to run other OS processes. Letting the thread sleep here allows the OS to run other processes when it will affect the game the least.

abstract class AbstractProcessWrapper and **class ProcessWrapper(Process, Policy)**: The former is a parent class providing unified API, the latter wraps the `runProcess()` function generated by `EntityManager.registerProcess()`. The `Process` type parameter is the wrapped Process. Also provides run-time access to various information about `Process`, such as performance diagnostics, name, and so on.

Game state

struct GameState(Policy) handles game state storage. Two instances exist (one for past, one for future state) and are swapped between game updates. The `Policy` parameter is the same parameter passed to `EntityManager`. Simplified layout of `GameState`:

```
// maxComponentTypes! adds the number of builtin component types to
// Policy.maxUserComponentTypes
ComponentTypeState!Policy[maxComponentTypes!Policy] components;
// Internally, and Entity is just a 32-bit unsigned integer

Entity[] entities;
```

struct ComponentTypeState(Policy) stores components of one type as well as any data used when processing those components, e.g. component counts per entity:

```
// A custom, "unsafe" fast array container; plain array could be used as well
ComponentBuffer!Policy buffer;
// Number of components of this type in each entity.
Policy.ComponentCount[] counts;

bool enabled;
```

`buffer` represents $components_i$ from section 2.2; `ComponentTypeState.counts` represents $componentCounts_i$. `enabled` is needed because we use a fixed-size array of `ComponentTypeState!Policy`, and not all items of that array may be used.

Between game updates, component data from the *former past* `GameState` instance is discarded, while data of the *former future* instance is kept as it becomes *new past*. The basic implementation is simple, although we use some optimizations to improve speed; see source code for details. `GameState.entities` and the `ComponentBuffer!Policy` instance used to store the builtin `LifeComponent` are sequentially traversed, only copying alive entity IDs to *new* future state. *New past* state still contains entity IDs that died in the previous game update to keep alignment between entity IDs and components, as components generated for dead entities in previous update still exist.

struct EntityRange(EntityManager, Process), instantiated once per Process, implements the low-level details of entity iteration for that Process (the `Process` type parameter), including Process-entity matching. It behaves as an input range [3] over all entities, even those that do not match `Process`. Internally `EntityRange` stores pointers to past component data of component types read by `Process.process`.

Scheduling

class Scheduler encapsulates both scheduling and time estimation.

abstract class SchedulingAlgorithm is defines API for scheduling algorithms. The most important method of `SchedulingAlgorithm` is:

- `protected Flag!"approximate" doScheduling(TimeEstimator estimator)`: uses execution times from `estimator` to assign Processes to threads, returns a flag specifying whether the schedule generated is approximate or optimal.

abstract class LPTSchedulingAlgorithm is a `SchedulingAlgorithm` implementation based on the LPT algorithm. Its `doScheduling` implementation:

Function:

```
override Flag!"approximate" doScheduling(TimeEstimator estimator) @trusted nothrow
{
    // Change procIndex_ length if needed (if Process count changes at runtime)
    while(procIndex_.length < procInfo_.length) { procIndex_.put(0); }
    procIndex_.length = procInfo_.length;

    // Get estimated execution time duration of specified Process from estimator
    ulong duration(ref ProcessInfo info) nothrow @nogc
    {
        return estimator.processDuration(info.processIdx);
    }

    // Initialize procIndex_ as array of indices to procInfo_ contents sorted by Process duration.
    // This is sorted in reverse order (shortest to longest - we then process the index in reverse.)
    procInfo_[].makeIndex!((l, r) => duration(l) < duration(r))(procIndex_[])
               .assumeWontThrow;
    auto index = procIndex_[];
    // Assert that we sorted it right.
    assert(duration(procInfo_[index.back]) >= duration(procInfo_[index.front]),
           "Index not sorted from shorter to greater duration");

    // Assign processes to threads, from slowest to fastest, always to the thread with smallest load
    while(!index.empty)
    {
        const longest = index.back;
        const dur = duration(procInfo_[longest]);
        // minPos gets the range starting at position of the least used thread
        const threadIndex = threadUsage_.length - threadUsage_.minPos.length;
        threadUsage_[threadIndex] += dur;
        procInfo_[longest].assignedThread = cast(uint)threadIndex;
        index.popBack();
    }

    return Yes.approximate;

}
```

Due to details of our implementation, this code appears to differ significantly from LPT pseudocode in section 3.2 , but is in principle same, with the following notes:

- We refer to Processes through an external index array, which is sorted and traversed in reverse order
- Most importantly, we use an array instead of priority queue of threads for simplicity (both in implementation, and to avoid even more memory indirection overhead). This means the complexity of the implementation is $O(n^2)$ rather than $O(n \cdot log(n))$; in our benchmarks this does not have a measurable effect, although extreme cases (~1000 Processes) may need a priority queue-based implementation.

abstract class COMBINESchedulingAlgorithm implements the COMBINE scheduling algorithm. Its code is can be found in `source/tharsis/entity/scheduler.d`.

abstract class TimeEstimator defines API for time estimation. Its most important members are:

- `final ulong processDuration(size_t processIdx) @trusted pure nothrow const @nogc`: Gets estimated execution time duration for specified Process in the next update.
- `void updateEstimates(const ProcessDiagnostics[] processes) @trusted nothrow`: Run the time estimation algorithm with diagnostics data (measured during the previous update) as input.

final class StepTimeEstimator: TimeEstimator is a time estimator implementation based on the algorithm detailed in section 3.2 . Its `updateEstimates` implementation follows:

Function:

```
override void updateEstimates(const ProcessDiagnostics[] processes) @trusted nothrow
{
    assert(estimateFalloff_ >= 0.0 && estimateFalloff_ <= 1.0,
           "estimateFalloff_ must be between 0 and 1");
    const procCount = processes.length;

    // Compensate for possible changes in Process count
    while(timeEstimates_.length < procCount) { timeEstimates_.put(0); }
    timeEstimates_.length = procCount;

    // Update time estimates for all Processes.
    foreach(id, ref proc; processes)
    {
        const prevEst = timeEstimates_[id];
        const duration = proc.duration;
        // If process ran longer than estimated, increase the estimate to duration.
        if(duration >= prevEst)
        {
            timeEstimates_[id] = duration;
            continue;
        }

        // If the process ran faster than estimated, lower the next estimate based on
        // estimateFalloff_.
        const diff = prevEst - duration;
        timeEstimates_[id] = prevEst - cast(ulong)(diff * estimateFalloff_);
    }

}
```

This implementation is almost identical to pseudocode from section 3.2 with the exception of handling cases when Process count changes.

4.2 Test game

The game we use for testing and benchmarking our ECS is not an proper *game*; it has no goals and the "mechanics" are very basic. The game world contains movable entities drawn from a dimetric (often incorrectly called "isometric" in the game development community) point of view, similarly to many RTS and RPG games.

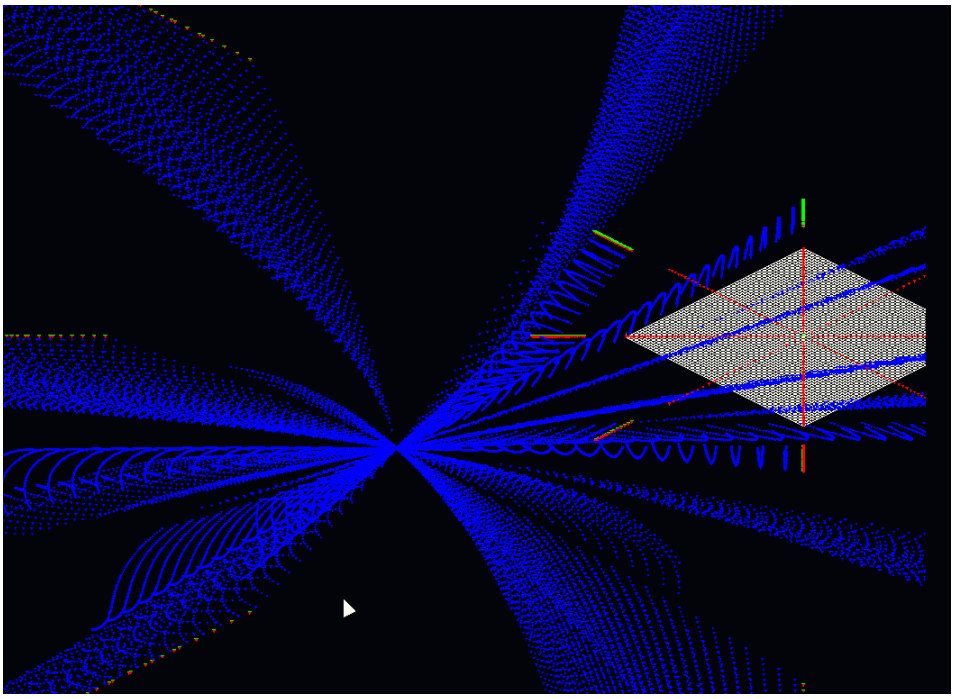

Figure 4.1: The test game. The various colored objects are entities simulated in the game.

Multiple entities can be selected by dragging the mouse and ordered to move to or fire their weapons at a location by clicking. "Firing their weapons" produces projectiles, which are also entities.

The open source SDL2[6] library is used for window management, user input, and other OS abstractions, while the OpenGL[7] API is used for graphics. Some code from the GFM[8] public domain library has been used as well.

Various benchmarking functionality is built into the game, including the ability to enable/disable graphics output (to see the effect of the GPU driver on multi-threaded performance), and to record/replay player input (used to ensure that user input does not change between benchmark runs).

Detailed controls of the game are described in its `README.html` file.

[6] http://libsdl.org

[7] http://d-gamedev-team.github.io/gfm/

[8] http://opengl.org

5 Benchmarking, Results

5.1 Test environment

We used two machines (both double-socket) for benchmarking:

- `AMD`: **2x8-core AMD Opteron 6134** (Magny-Cours) at **2.3 GHz** (128k*8 L1, 512k*8 L2, 6M*2 L3) with 24 GiB of 1333 MHz DDR3 RAM
- `Intel`: **2x8-core Intel Xeon E5-2650L** (Sandy Bridge) at **1.8 GHz** (64k*8 L1, 256k*8 L2, 20M*1 L3) with 128 GiB of 1600 MHz DDR3 RAM

Both machines ran **Ubuntu 14.04.2 LTS** with no grahical environment (no X server).

5.2 Tested cases

We've tested the performance of our implementation along "dimensions" that can be seen below, with test runs for each combination of values (e.g `few-small-constant-1`):

> **Legend**
>
> In `few`: `[1, 2, 2, 3, 4]`, the array means that generated (see further below) Processes are distributed so that for every 5 Processes, one has 1 past component parameter, two have 2, one has 3 and one 4 such parameters. Similar for **overhead size** (relative size of overhead). For **overhead patterns**, the array contains *patterns* instead of *numbers*, so e.g. `[[1], [1, 2]]` means that half of the generated Processes have constant overhead (always 1) and half have a [1, 2] pattern, where every other frame has double overhead.

Number of components read per Process (**number of past component parameters** for the `process()` function):

- `few`: `[1, 2, 2, 3, 4]`
- `many`: `[5, 5, 6, 7, 8]`
- `mixed`: `[1, 2, 3, 4, 5, 6, 7, 8]`

Process execution **overhead size** (time a Process takes to process an entity):

- `small`: `[1]`
- `large`: `[44, 46, 47, 48, 50, 50, 50]`
- `mixed`: `[1, 3, 5, 7, 9, 11, 13, 15, 17, 25, 50]`

Process execution **overhead pattern** (how does time spent by a Process to process an entity change over time - between game updates)

- `constant`: `[[1]]`
- `regular`: `[[1.0, 1.0, 1.0, 2.0], [0.5, 1.0, 1.5, 1.0], [0.9, 1.0, 1.1, 0.9, 10.0], [1.0, 2.0, 1.0, 0.5, 1.0, 0.5]]`
- `irregular`: 10 randomly generated patterns 1000 to 5000 game updates long; every number in each pattern has a 95% probability of being uniformly distributed in $[0.33, 3.0]$, and a 5% probability of being uniformly distributed in $[0.1, 10]$ (this simulates large overhead spikes).
- `mixed`: `[[1], [1.0, 2.0, 1.0, 0.5, 1.0, 0.5], [` irregular pattern `]]`

Number of threads:

- `1` to `15`, we always left one of the 16 cores free to avoid interfering with OS overhead too much

A separate binary was generated using the `ldc2 0.15.0-beta1` compiler with flags `-release -enable-inlining -disable-boundscheck -O` for every combination of listed past component parameter count, overhead size and overhead pattern. Each of these binaries was then run multiple times to cover thread counts from 1 to 15. The runs were replaying user input from demo file `demo_main-benchmark.yaml`, which can be found on attached DVD. Over a run of this demo, new entities are gradually being added, with entity count increasing to about ~41000 entities; also increasing the workload of the ECS, increasing game update times and so on.

To simulate overhead of a complex game we used generated Processes; the arrays/patterns listed above were parameters for Process code generation in each build.

The actual number of Processes did not change between tests; there was a fixed number (`12`) of hand-written Processes needed for the test game to function, with a larger number (`64`) of artificial overhead Processes.

5.3 Test results

Values shown in the charts and tables below are averages, maximums, etc. over a run (which could take anywhere between 1 to 20 minutes). For example, a game update duration data point in a chart represents an average for e.g. `many-large` (shown in the chart title) `-mixed` (one of the graphs in the chart) for `5` threads (`x` coordinate in the chart).

The sheer size of generated data is too large to list here; we show the charts with most interesting data here. More charts can be found on attached DVD.

Chart explanation

Every chart consists of 5 vertical subcharts. Explanation of these subcharts, from left to right:

1. Graphs `constant`, `regular`, etc. show average update duration for the named overhead patterns. Graphs with names ending with `-max` show maximum (lag) game update time over the run, and those ending with `-lim` show a bottom bound on possible game update time calculated as the greater of total Process time divided by thread count and slowest Process duration. Even with perfect scheduling and time estimation, update can not be faster than the slowest Process or total Process time equally divided among threads.

   Shows how much faster the game gets with more cores

2. A sum of Process execution time from all threads (code that runs in threads **without synchronization**)

   Shows how saturating cores, caches, etc. affects performance even without direct synchronization overhead

3. Average time estimation error and - in graphs with names ending by `-uest` - average time estimation *underestimation*; if the latter is negative, the estimator has a tendency to *overestimate* rather than underestimate, which is our intention (see section 3.2). This is an average of averages (for each update, estimation errors for all Processes are averaged to get the average error for the update, and these are then averaged together).

Shows how much margin for improvement there is in time estimation

4. Number of entities that simultaneously exist in the simulation before the frame duration average over a 1 second sliding window reaches 33 ms (30 FPS) (remember - during each benchmark run, the number of entities (i.e. overhead) gradually increases). Due to the tendency of this value to deviate, an *average* graph is added, which averages results between `constant`, `regular`, `irregular` and `mixed` runs. As the maximum number of entities in our tests was around 41000, any potentially higher results are clamped to this value.

   Shows how adding more cores allows the game to use more entities

5. Average deviation of game update duration over the course of a 1 second sliding window. Where avg is the average game update duration over the 1 second sliding window, graphs suffixed with `-dev` show the deviation, **defined as** $|duration - avg|/avg$, graphs suffixed with `-jtr` show the percentage of "jitter" updates, defined as updates $duration > avg \cdot 2$, and the graphs suffixed with `-lag` show the percentage of "lags", similar to "jitter" but $duration > avg \cdot 5$.

   Shows how smooth the framerate can be with different core counts

The charts shown here are from measurements with `many` Process parameters and `large` Process execution overhead, where overhead of actual work done in the ECS is the largest and where frame time could almost always reach 33 ms while running the demo (for measuring how entity count possible with 30 FPS scales with thread count). For other measurements, see attached DVD.

Note that all shown charts show measurements using the LPT algorithm for scheduling. We also made measurements with the COMBINE algorithm, but they have shown to be nearly identical to results with LPT; it appears LPT is "good enough" for at least the cases our measurements cover.

many-large on Intel without hyper-threading/frequency scaling

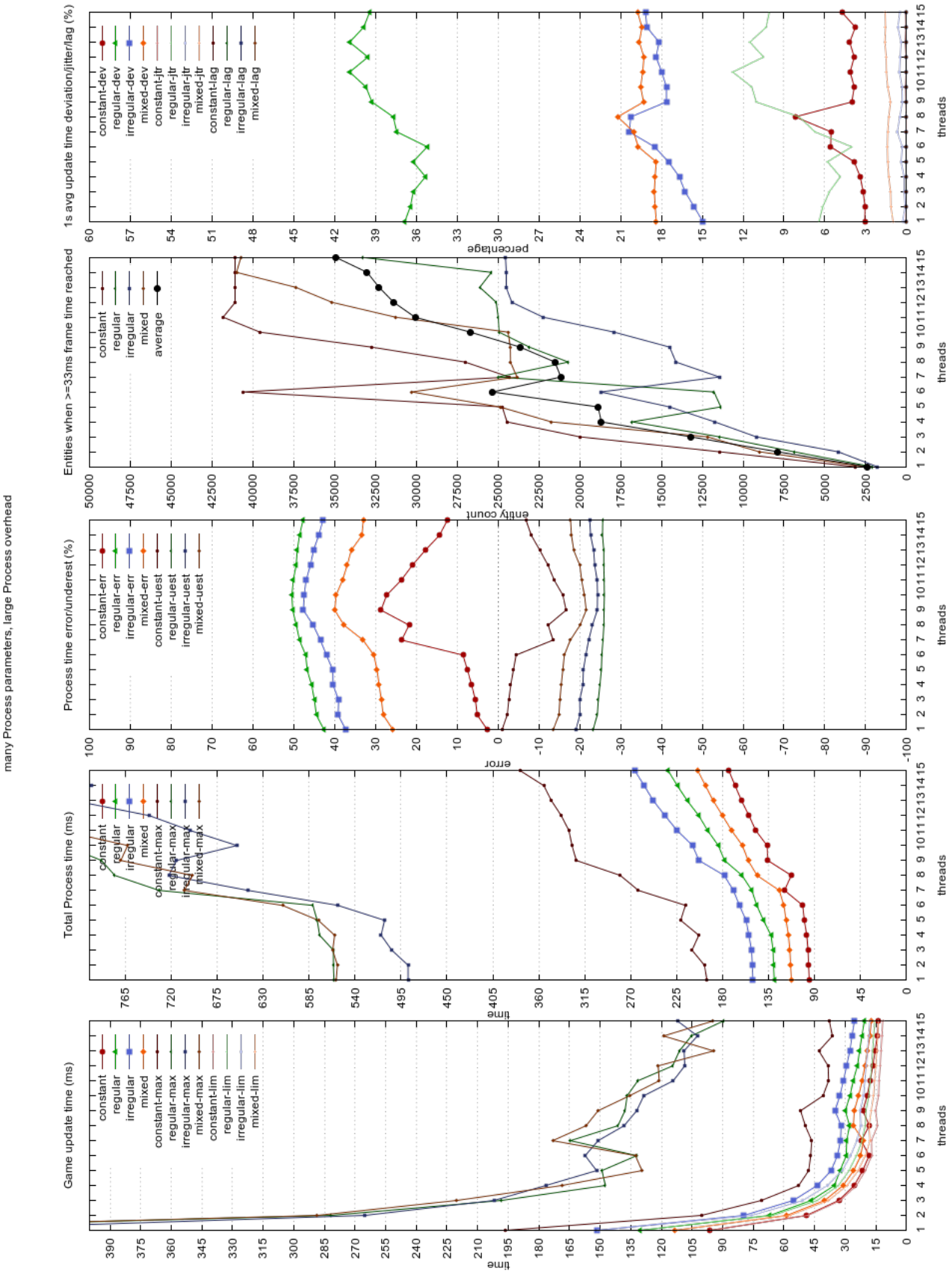

Figure 5.1: Performance for many past component parameters, large Process overhead on Intel with **no hyper-threading** and **no frequency scaling disabled**. See attached DVD for tables with exact values.

Notes

Game update time improves well with thread count up to around 8 threads. Once both CPUs are used, update time spikes and the continues to scale at a slower rate. Also, it seems (from the `-max` graphs) that saturating one CPU leads to larger overhead spikes.

Intuitively, sum of time in (unsynchronized) Process code for all threads (**total Process time**) should not change (it is always the same amount of work), but it can be seen that it increases with thread count, especially when using two CPUs. This might be caused by saturation of resources shared by CPUs/cores (e.g. shared caches, memory bus). This limits scaling of our ECS as update time can never be better than total process time divided by thread count.

As intended, the **time estimator** has a tendency to overestimate rather than underestimate Process execution times, but the error magnitude still leaves a margin for improvement in future work. Also, iterestingly, it seems our time estimator generates worse results for `regular` than `irregular`.

Number of entities with a <33 ms update time appears to scale very well with low thread counts, with some slowdown once two CPUs are used.

Game update time deviation for `regular` is greater than for other overhead patterns. This may mean our time estimator does not handle these patterns well. Jitter is uncommon but may be an issue with `regular` on larger thread counts, especially with HT disabled. There is almost no lag (the `-lag` graphs stay around zero).

many-large on AMD

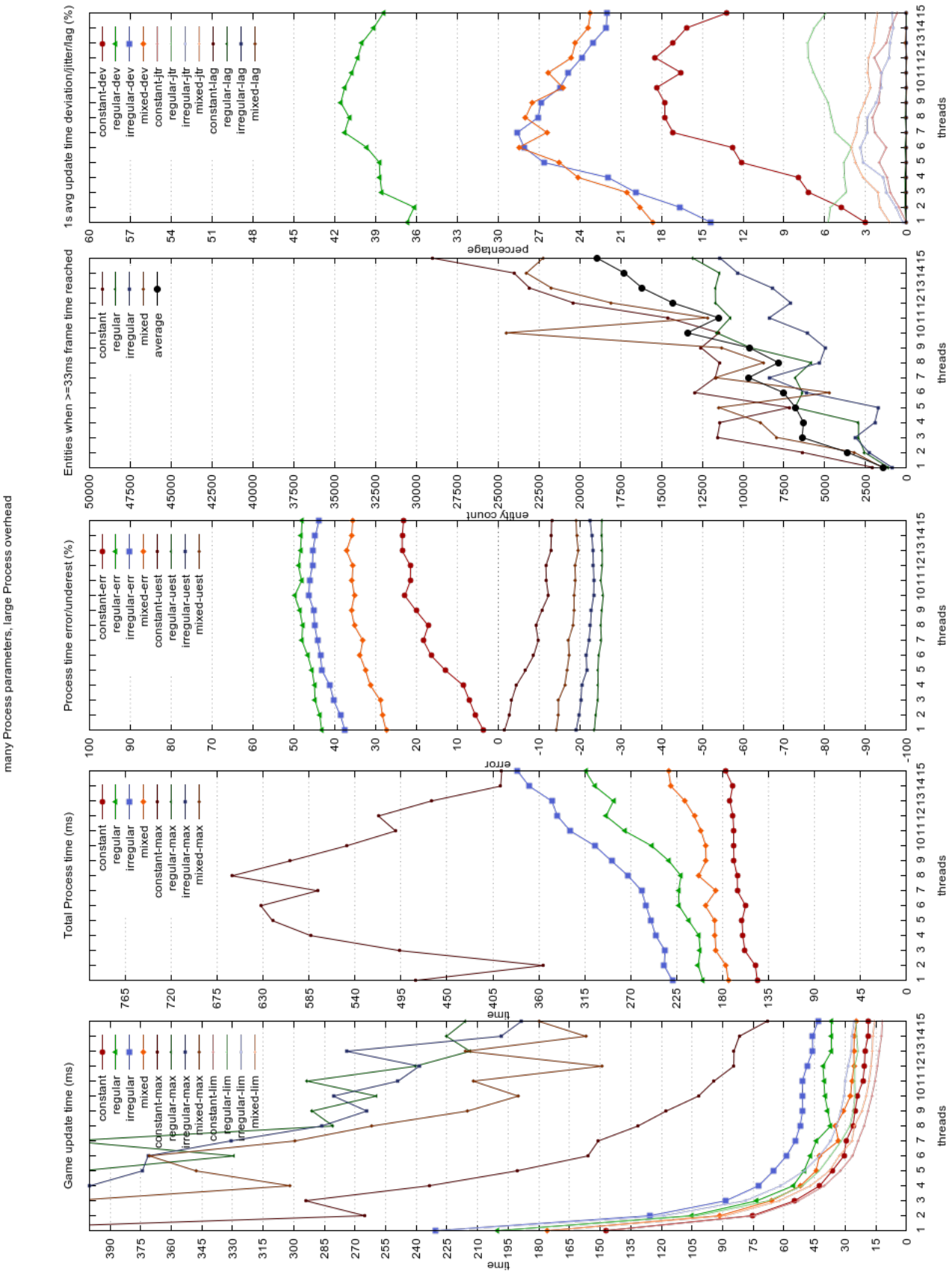

Figure 5.2: Performance for `many` past component parameters, `large` Process overhead on `AMD`. See attached DVD for tables with exact values.

Differences from Intel

Note that `AMD` has no equivalent to hyper-threading. We were unable to determine whether frequency scaling is being used by this CPU. Also, the Magny-Cours architecture is older than Sandy Bridge so slower performance is to be expected.

- With large thread counts, **update time** with `regular` and `irregular` overhead patterns does not scale as well on `AMD` as on `Intel`. On the other hand, `constant` seems to scale better. A similar effect can be seen in **total Process time** graphs, where the time for `constant` increases slower compared to `Intel` while e.g. `irregular` increases even faster.
- **Process time estimation** errors seem to be somewhere in between the `Intel` HT/no-HT results.
- **Number of entities with a <33 ms update time** seems to scale regularly with thread count, although it is heavily affected by overhead patterns.
- **Game update time deviation** indicates less stable framerate compared to `Intel`, which interestingly improves when thread count comes close to 15. There is also more jitter with overhead patterns *other* than `regular`, although it's not large enough to be a problem.

Conclusion

We have designed an ECS architecture that separates past from future game state, building on ideas from existing component-entity designs summarized in the first chapter. We have described the memory layout and execution in our ECS architecture.

We are using a fairly simple organization of component data into tightly packed per-type buffers with immutable past and mutable future versions. This allowed us to execute ECS Processes (Systems) in a straightforward sequential loop, minimizing cache misses and branch mispredictions. Our design can be efficiently implemented in programming languages supporting compile-time code generation.

In Chapter 3, we have proposed a parallelized ECS, distributing Processes among multiple thread without need for manual management. We have detailed the problems of scheduling Processes to threads and estimating Process execution times, and applied multiple existing algorithms to the former as well as a simple algorithm to the latter.

On the implementation side, we have a working implementation of the our ECS design written in the D programming language and a simple game, which we used to benchmark this implementation on two different many-core systems.

Benchmark results show that our ECS scales well with small core counts, and scales to large core counts (>8) to some degree, but there is still margin for improvement. While Process scheduling seems to be a "solved problem" with currently used approximate algorithms, our algorithm for Process execution time estimation is far from perfect. However, we believe the ECS is applicable to game development on most modern gaming platforms even in its current state; further improvement will make it more future-proof, as core counts in gaming hardware increase.

In future work, we hope to explore different algorithms for Process execution time estimation, possibly making use of statistics about a Process collected at run-time or programmer-specified hints at compile-time, and to further extend the ECS design for both performance and usability, e.g. by allowing non-critical Processes that may not run on every game update, component types supporting multiple components per entity, and "out-of-band" access to past components of entities other than the one being currently processed.

Bibliography

[1] Tony Albrecht. Pitfalls of Object Oriented Programming. *Game Connect: Asia Pacific*. 2009.

[2] Andrei Alexandrescu. Modern C++ Design: Generic Programming and Design Patterns Applied. *Addison-Wesley Professional*. 2001.

[3] Andrei Alexandrescu. On Iteration. [online] [Retrieved 2015-04-10] Available on the internet: http://www.informit.com/articles/printerfriendly/1407357

[4] Michael A. Carr-Robb-John. The Game Entity. *Game Developer Magazine*. November 2011.

[5] E.G. Coffman, Jr., M.R. Garey and D.S. Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*. 1978.

[6] Terrance Cohen. A Dynamic Component Architecture for High Performance. *Gameplay GDC Canada*. 2010.

[7] Tom Davies. Entity Systems. 2011. [online]. [Retrieved 2014-06-16] Available on the Internet: http://www.tomseysdavies.com/2011/01/23/entity-systems/

[8] R. L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal on Applied Mathematics*. 1969.

[9] Chung-Lee Lee, David Massey. Multiprocessor scheduling: combining LPT and MULTIFIT. *Discrete applied mathematics*. 1988.

[10] Noel Llopis. Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP). 2009. [online]. [Retrieved 2014-06-16] Available on the Internet: http://gamesfromwithin.com/data-oriented-design

[11] Richard Lord. What is an entity system framework for game development? 2012. [online]. [Retrieved 2014-06-16] Available on the Internet:
http://www.richardlord.net/blog/what-is-an-entity-framework

[12] Adam Martin. Entity Systems are the future of MMOG development. 2007. [online]. [Retrieved 2014-06-16] Available on the Internet:
http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/

[13] Chris Stoy. Game Object Component System. *Game Programming Gems 6*. 2006.

[14] Mick West. Evolve Your Hierarchy. 2007. [online]. [Retrieved 2014-06-16] Available on the Internet:
http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/

[15] Artemis, Artemis-odb [online] [Retrieved 2015-03-26] Available on the internet:
http://entity-systems.wikidot.com/artemis-entity-system-framework

[16] Ash entity framework [online] [Retrieved 2015-03-26] Available on the internet:
http://www.ashframework.org

[17] Concept (generic programming) [online] [Retrieved 2015-03-28] Available on the internet: http://en.wikipedia.org/wiki/Concept_(generic_programming)

[18] Crysis (computer game). Crytek. 2007.

[19] EntityX [online] [Retrieved 2015-03-26]
Available on the internet: https://github.com/alecthomas/entityx

[20] Entreri [online] [Retrieved 2015-03-26]
Available on the internet: https://bitbucket.org/mludwig/entreri

[21] ICE (computer game) [online] [Retrieved 2015-03-27]
Available on the internet: https://github.com/kiith-sa/ICE

[22] Multiprocessor Scheduling [online] [Retrieved 2015-04–04] Available on the internet: http://en.wikipedia.org/wiki/Multiprocessor_scheduling