

Parallelism in AI

Multithreading Strategies and Opportunities for Multi-core Architectures

Julien Hamaide

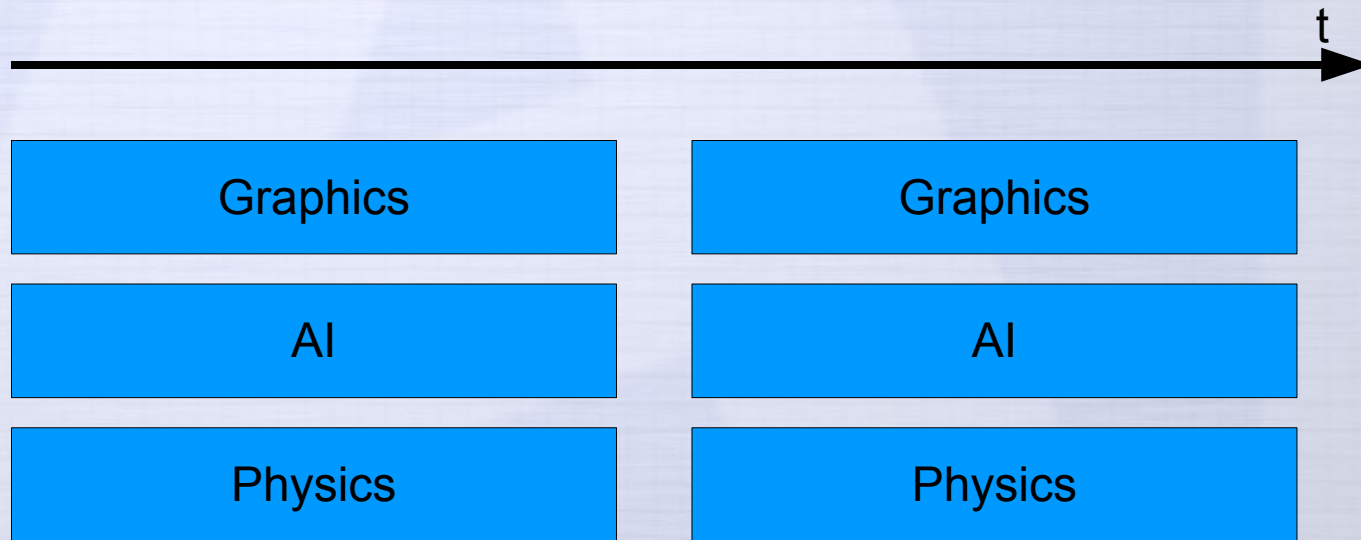


Parallelism in AI

- **Introduction**
- Data structures
- Code and Data usage
- High-level AI architecture
- Discussion

Multithreading

- Now :
 - Few cores
 - More tasks than CPU

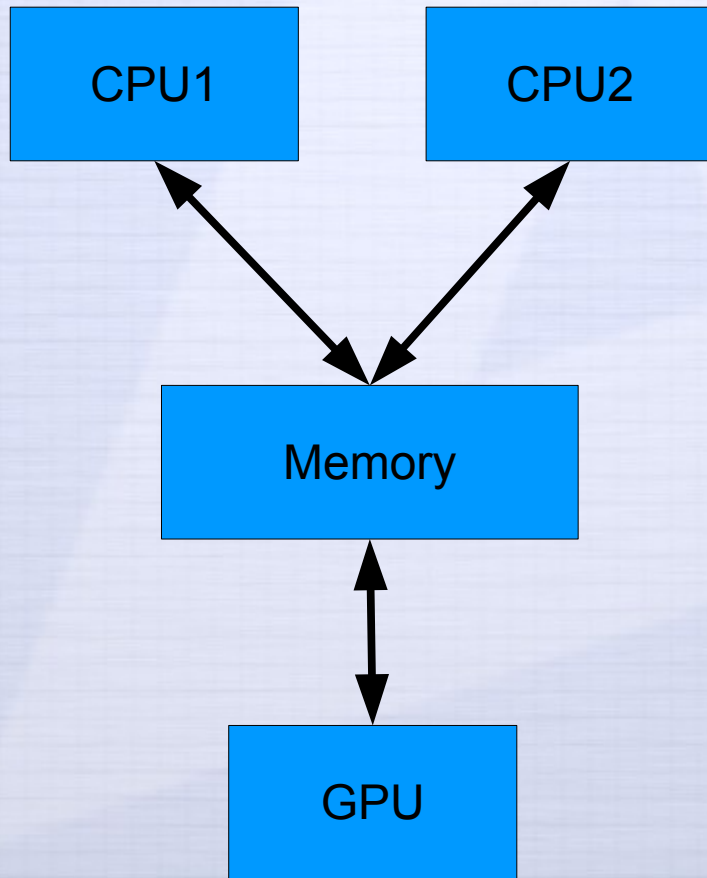


Multithreading

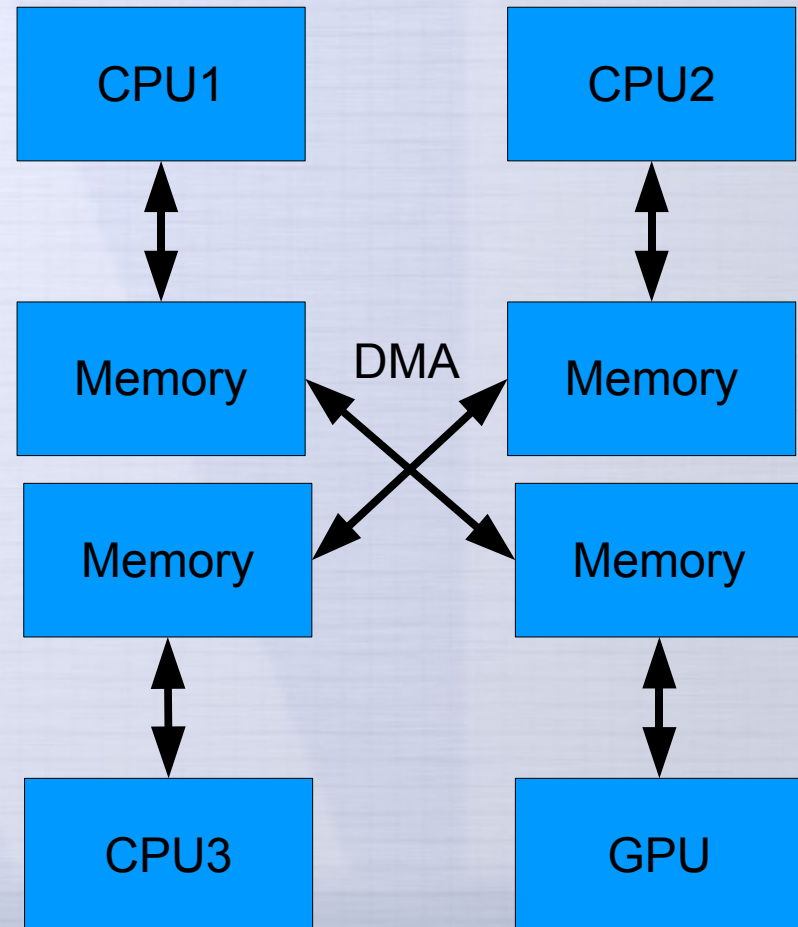
- In the near future
 - Hundreds or thousands of cores
 - How to split tasks in smaller pieces?
- Is it AI programmer's job?

Architectures

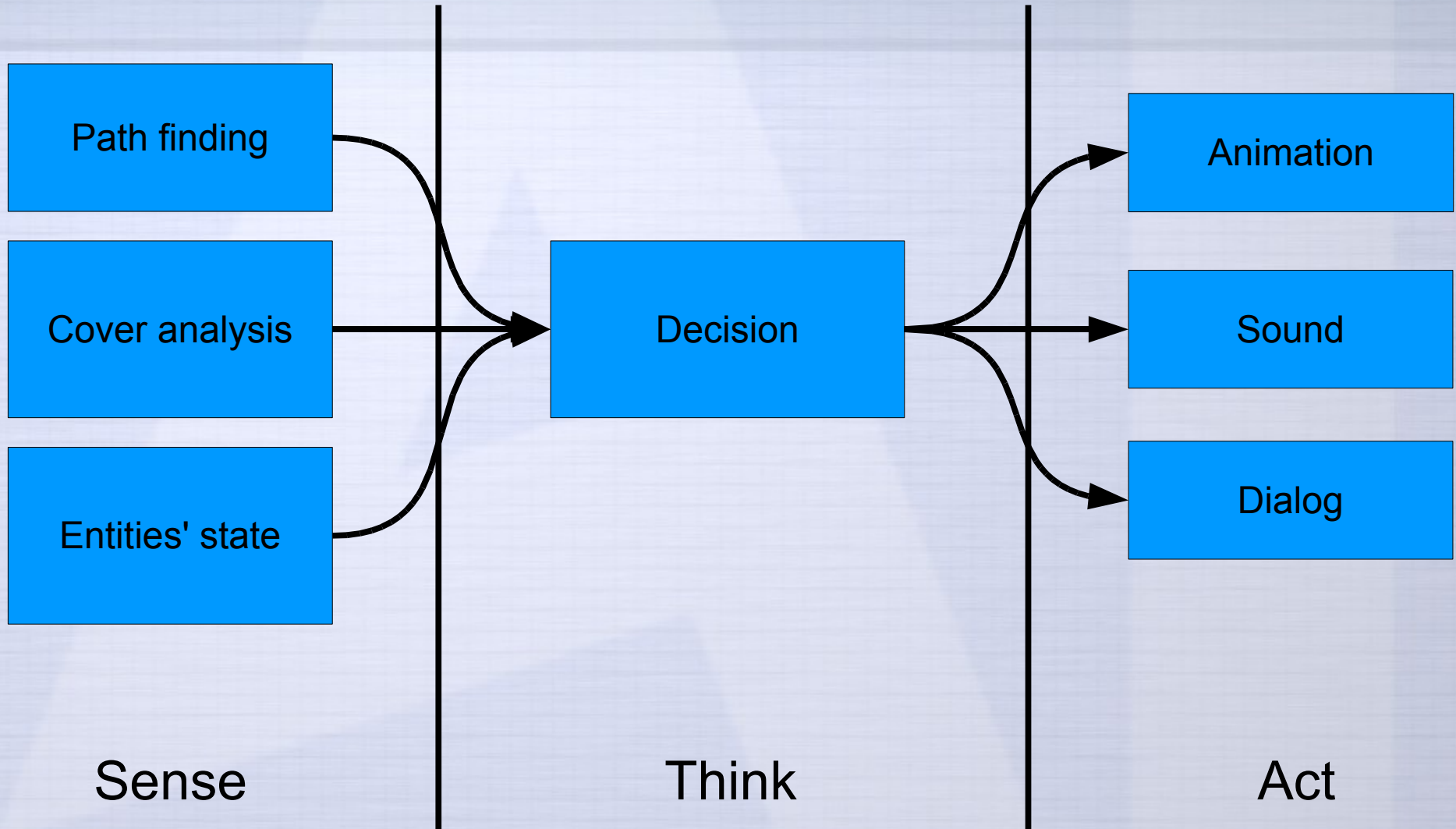
- Homogeneous



- Heterogeneous



AI Workflow



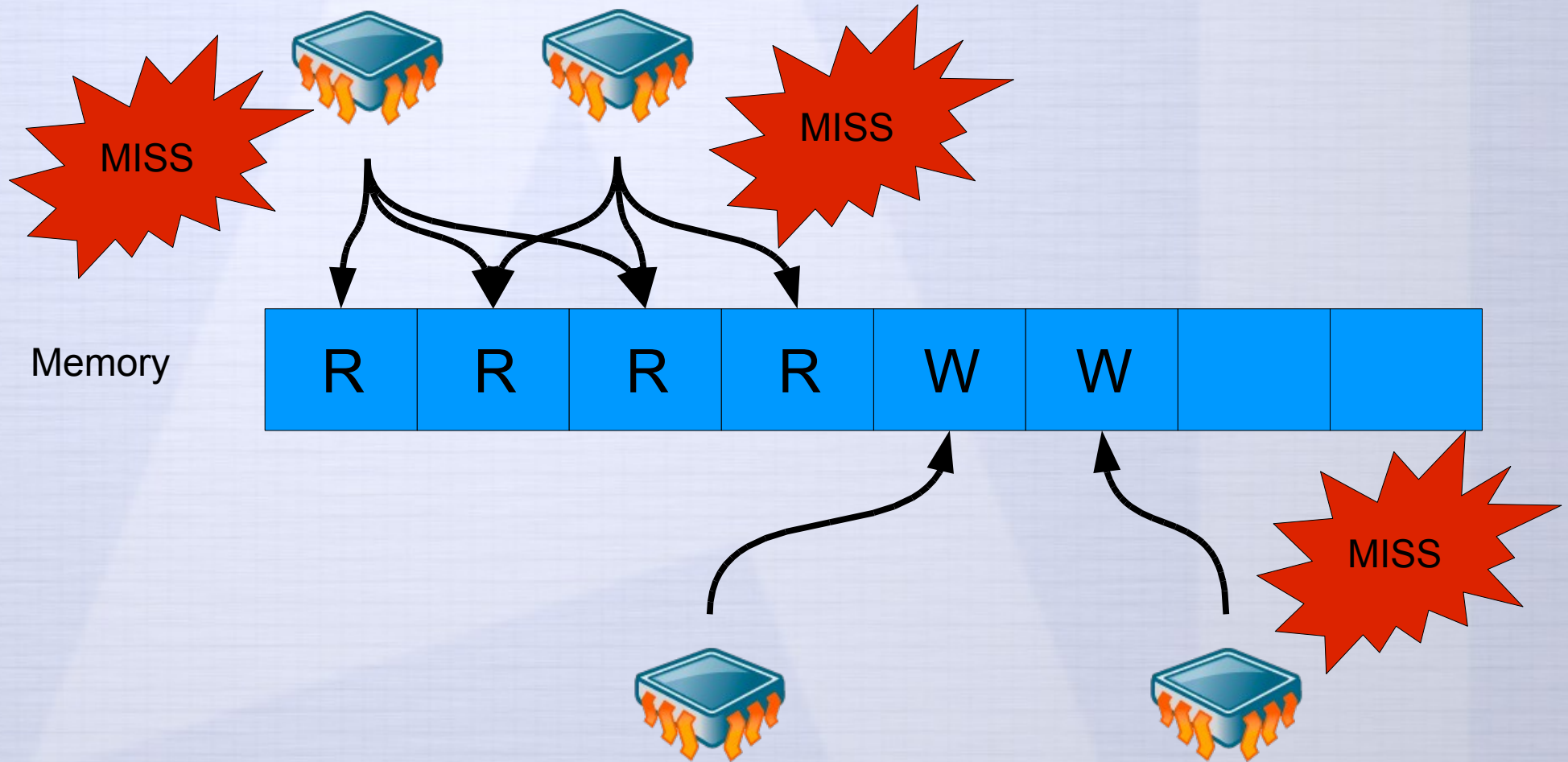
Parallelism in AI

- Introduction
- **Data structures**
- Code and Data usage
- High-level AI architecture
- Discussion

Cache coherence (1/2)

- Programming for multithreading is programming for cache first
 - Cache problems are worst in multithreading
 - Bus contention may appear if all CPUs access memory at the same time

Cache coherence (2/2)

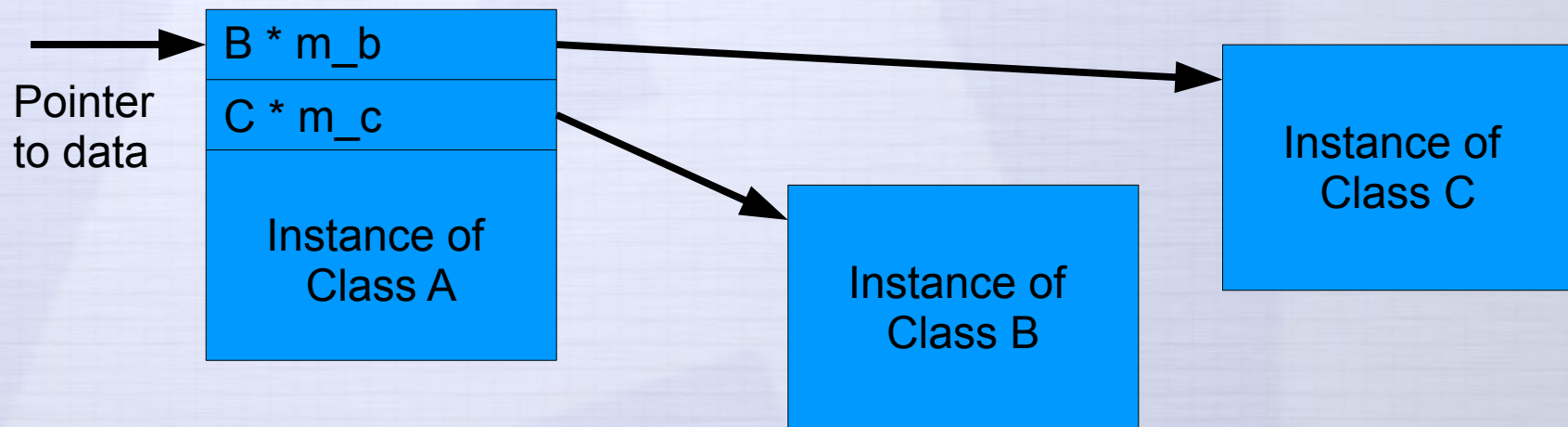


More advices

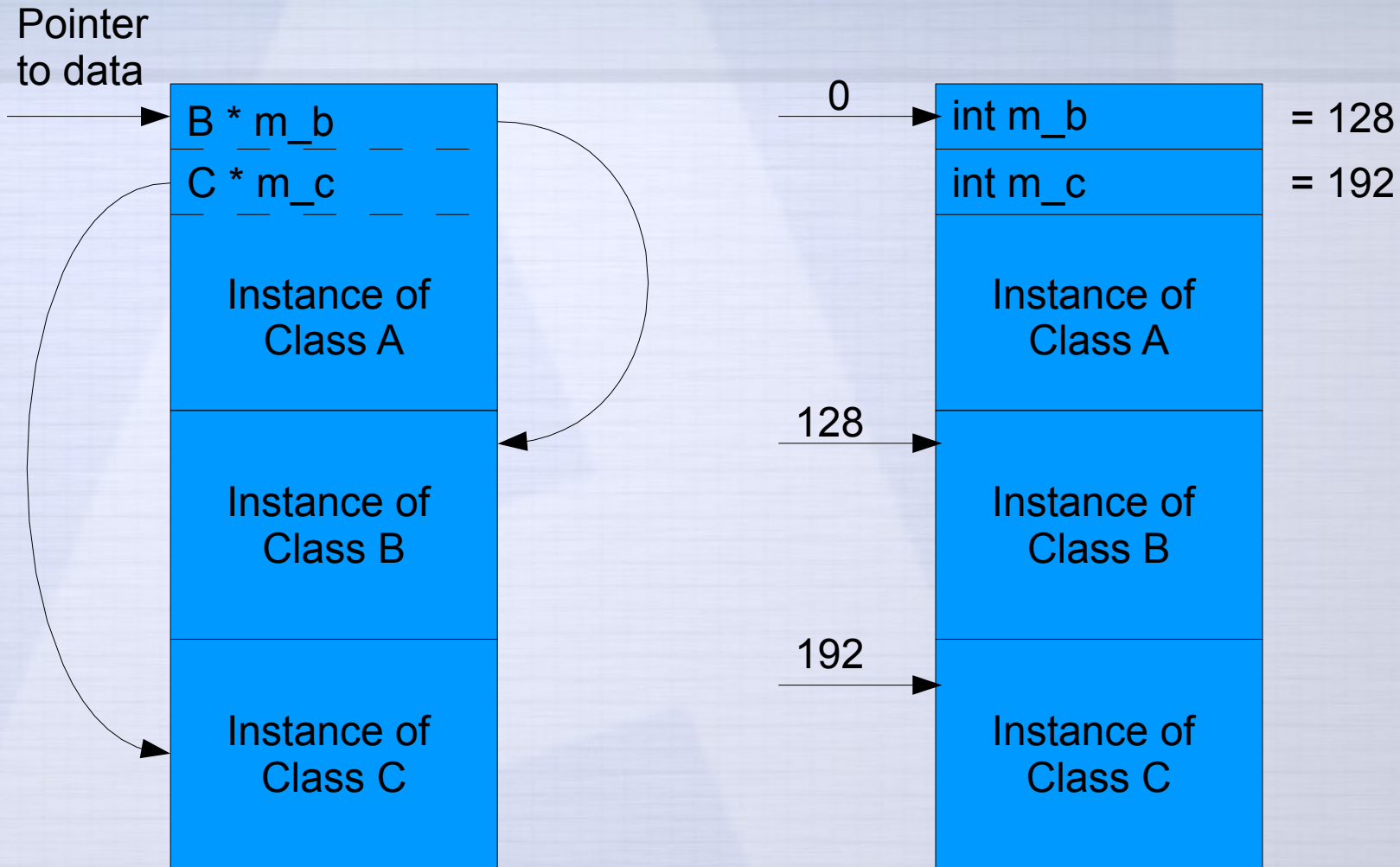
- Separate read from written data
- Keep data as small as possible
- Separate frequently read data from rarely read data
- Layout data in the same order they are read
- Align data to cache alignment

Position independency (1/5)

- Data must be position independent



Position independency (2/5)



Position independency (3/5)

- Data must be position independent
 - All objects in a big memory array
 - All objects are referenced relative to start array
 - Pointers become an integer and can be compressed to short
 - Short : 16 bits = 64K index
 - $64K * 4 \text{ bytes} = 256K\text{bytes}$ (SPU ;-)

Position independency (4/5)

- Advantages
 - Easy to send through DMA
 - No overhead of multiple allocation
 - Locality of data
 - Easy to serialize to and from file

Position independency (5/5)

- Helper to transform ID to pointer

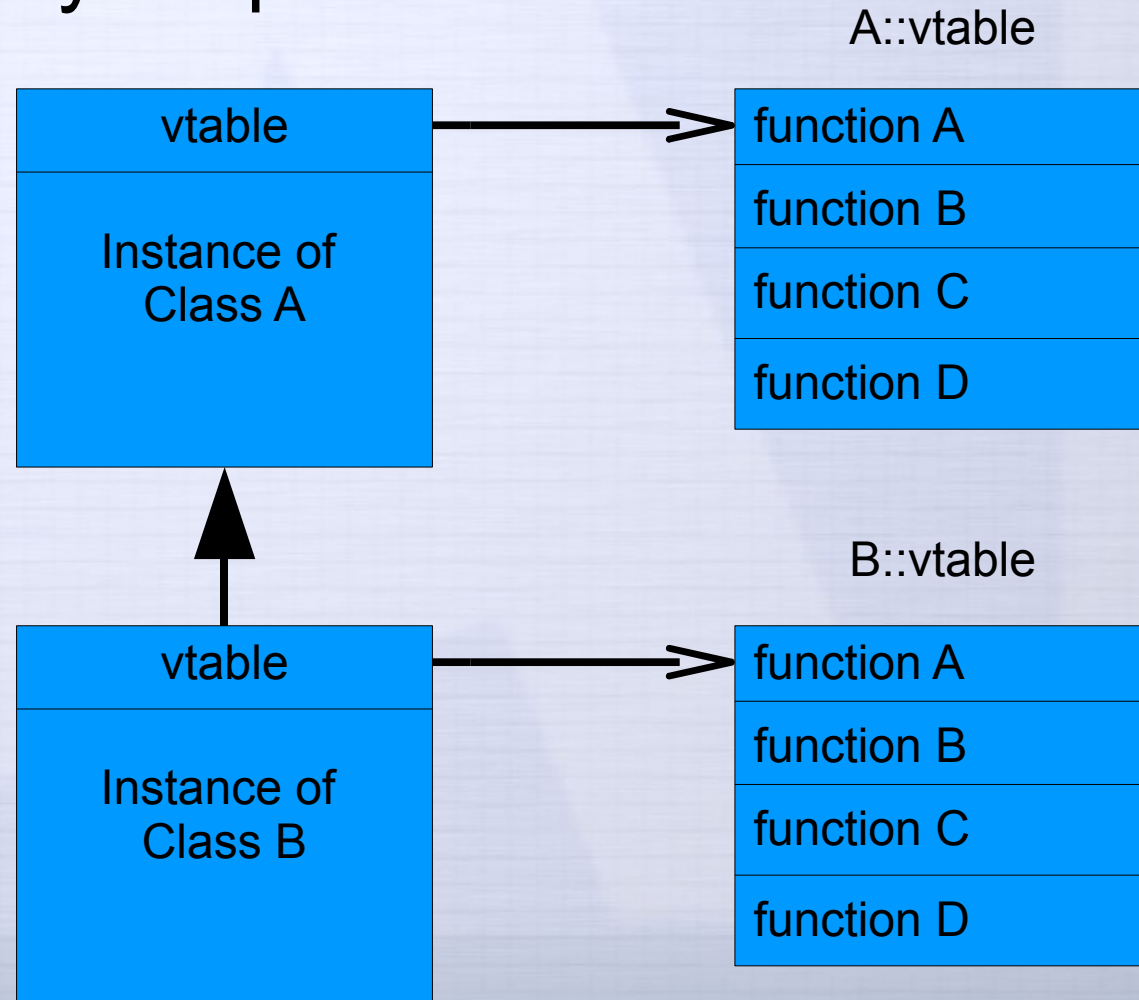
```
template< typename T> class Id
{
    T* operator() ( int * base_pointer )
    {
        return reinterpret_cast<T*>( base_pointer[ m_id ] );
    }
    int m_id;
}

class A
{
    Id<B> m_b;
    Id<C> m_c;

    void DoSomething() { m_b( base_pointer )->DoSomething(); }
}
```

Polymorphic class(1/2)

- Avoid polymorphic class



Polymorphic class (2/2)

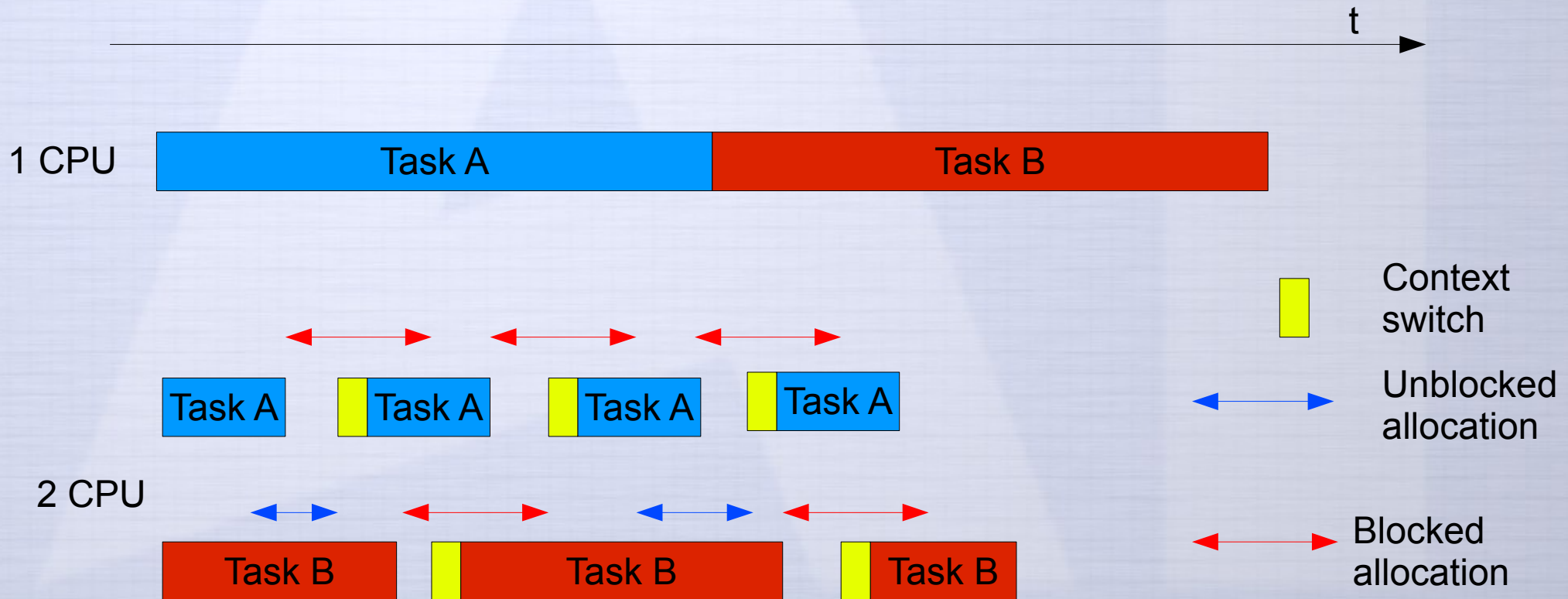
- Address of vtable is dependent on the executable.
 - It won't be valid after a DMA transfer
 - Can be patched, but incurs avoidable processing of data
- Vtable address is not local to data, it will surely raise a cache miss

Parallelism in AI

- Introduction
- Data structures
- **Code and Data usage**
- High-level AI architecture
- Discussion

Memory allocation (1/2)

- Avoid `::malloc` and `::new`
 - Implemented with locks



Memory allocation (2/2)

- Thread Local allocator
- Stack allocator easy to implement lock-free
 - Only an index to increment
- Pre allocate result memory
 - `vector.reserve()`;
 - Stack-based memory
 - `_alloca`

To lock or not (1/2)

- Lock-free algorithms are not easy to implement and debug
 - Heavily depends on architecture
- Does not always give better performance
 - First implement with lock
 - Try to decrease the number of contention points
 - Choose the best places to go lock-free

To lock or not (2/2)

- Some collections are simple and worth the try
 - Priority queue
 - Stack
 - Fifo list
- Try to keep locking equivalent collections
 - Validates the bug is not coming from lock-free collection
 - Replace with locking collection to debug

Parallelism in AI

- Introduction
- Data structures
- Code and Data usage
- **High-level AI architecture**
 - Double buffering
 - Message system
 - Job scheduling system
 - Asynchronous request system
- Discussion

Double Buffering (1/3)

- Parallelism via Double Buffering
 - All tasks run in parallel with each other with no dependency stall.
 - Each task can only read from previous frame's results
 - Each task can only write to itself

Double buffering (2/3)

- Only data read from outside must be double buffered.
- Data must remain cache friendly
 - Don't interlace read data with written data
 - Order data!!
- Switching buffers must be viewed as an atomic operation on all entities

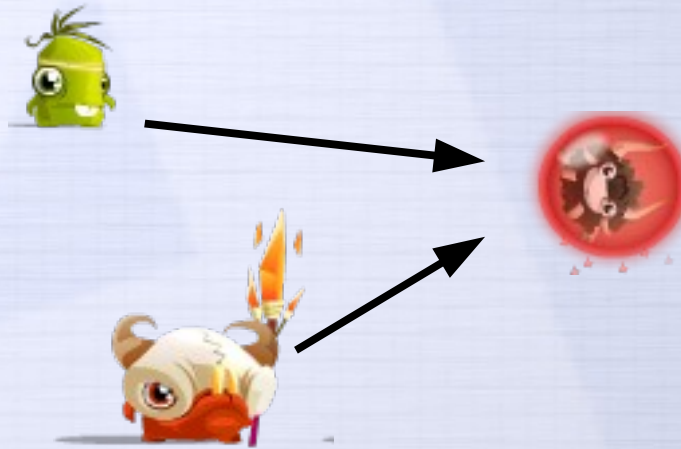
Double Buffering (3/3)

- Linearizability
 - In concurrent programming, an operation is linearizable, if it appears to take effect instantaneously, Wikipedia
- Double buffering provides a mean for linearizability by creating linearization points when switching buffers. All changes appear to take effect instantaneously.
- Enforce determinism
- Ease debugging

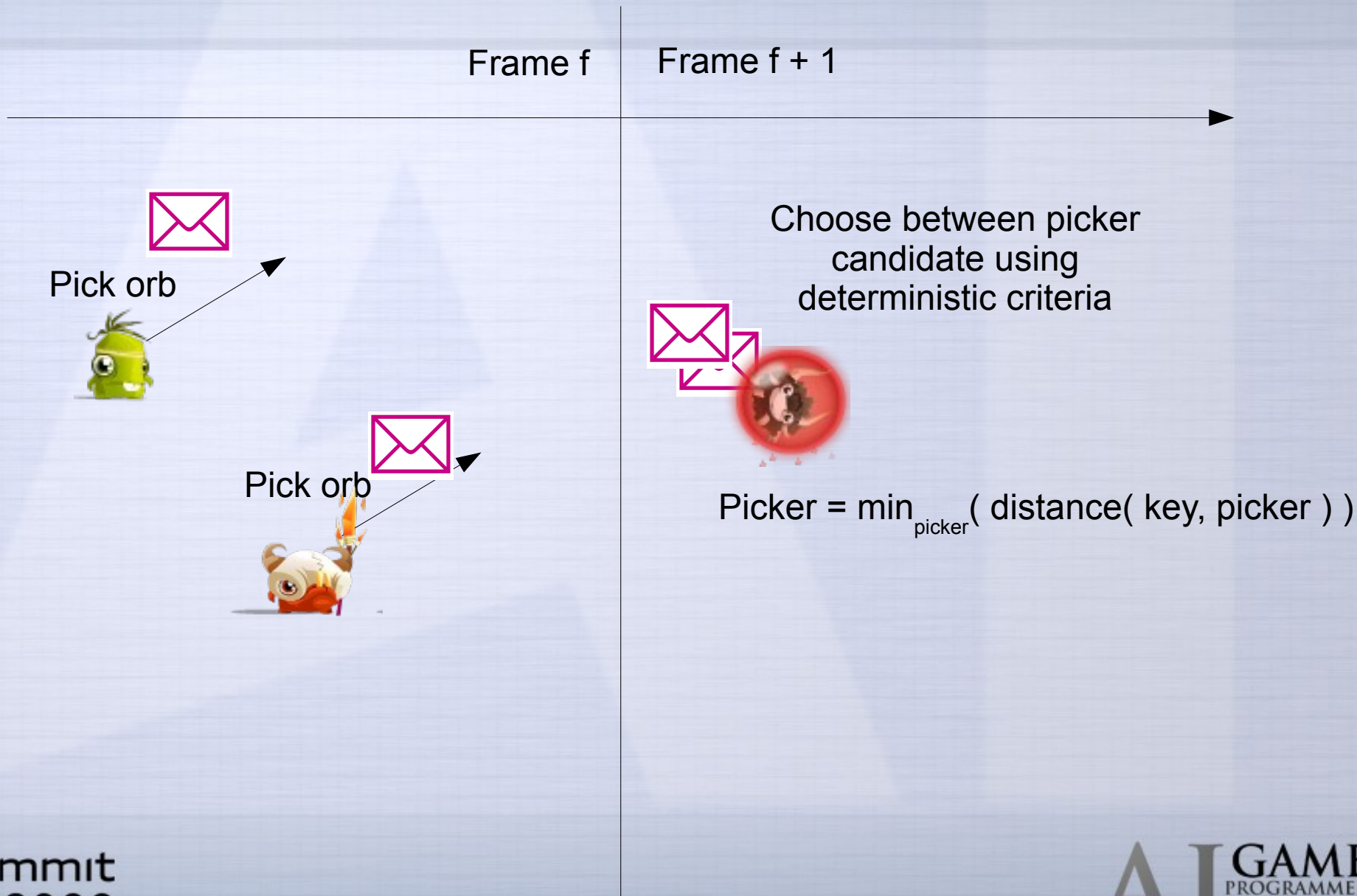
Message system (1/4)

- Messages are collected during the frame
- All messages are dispatched at the beginning of next frame
 - Synchronize all objects (linearization point)
 - Independent of sending order within a single frame
- Allows communication with other systems and entities without writing directly in their space

Message system (2/4)



Message system (3/4)



Message system (4/4)

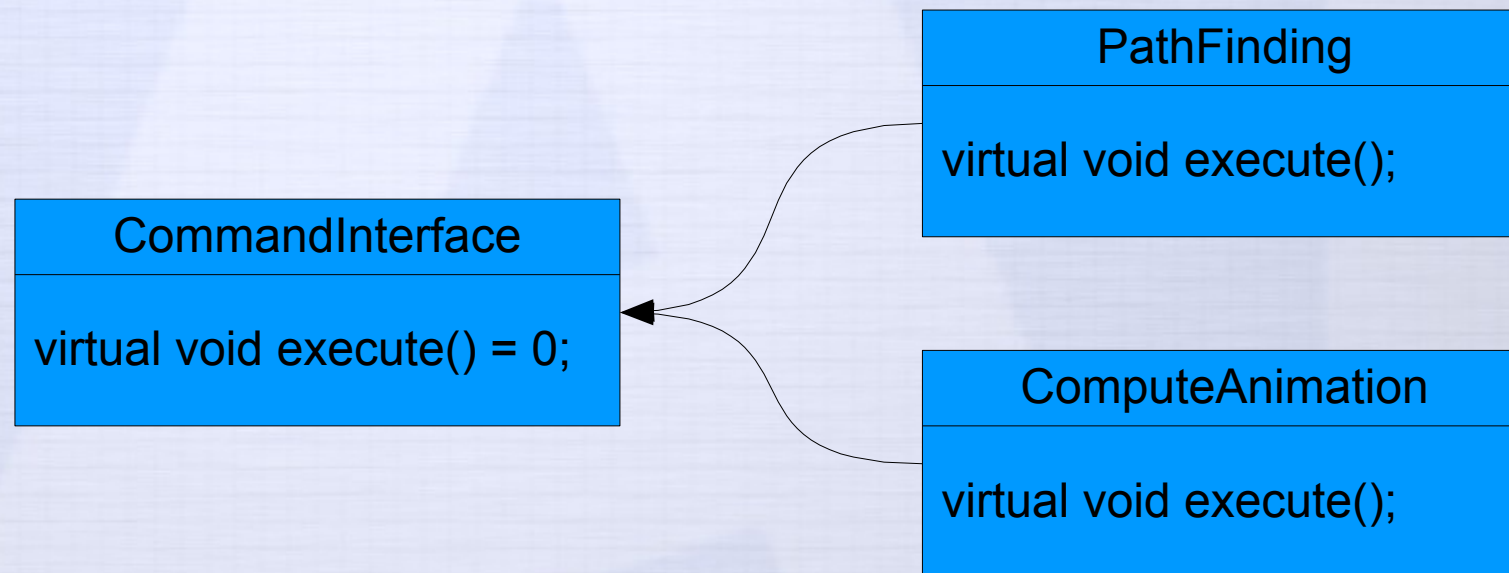
- Implementation
 - Lock-free linked list for insertion of sent message
 - Distribution of messages is made before objects' update
 - Only insertion must be thread-safe
 - All objects have a local list of sent message from the previous frame

Job Scheduling System (1/4)

- Command Pattern
- Execute command based on priority and dependency
- Might produce a « future » object

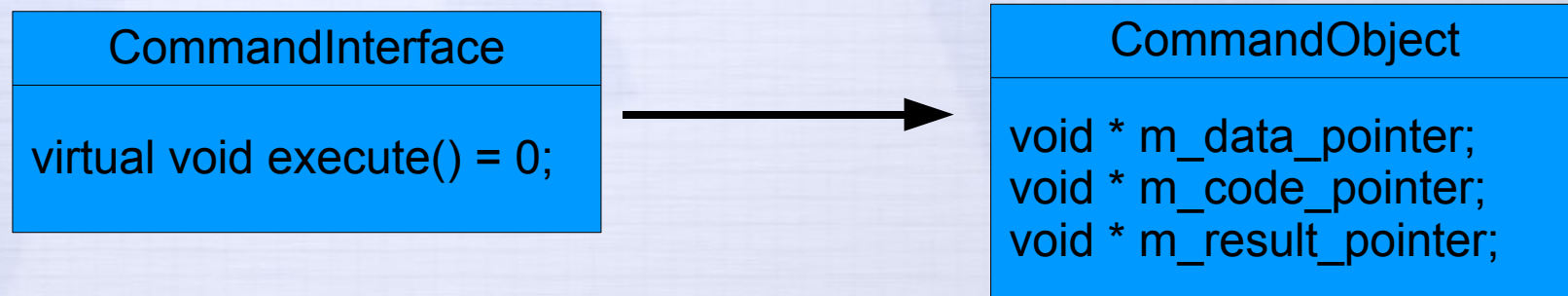
Job Scheduling System (2/4)

- Command Pattern

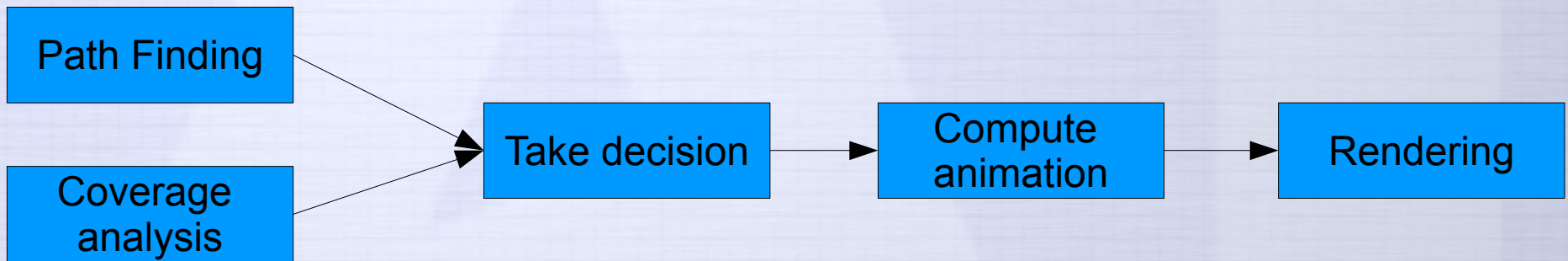


Job Scheduling System (3/4)

- Heterogeneous architecture



Job Scheduling System (4/4)



Future object (1/2)

- Response to a query that will be available in the future
 - No need to ask the job system about the state of the query
 - Can provide partial data (first segments of path, ...)

```
Future<Path> path = JobSystem::AddJob( new PathFindingQuery( ... ) );
```

```
If( path.IsAvailable() )  
{  
    // use path as a Path object  
}
```

Future object (2/2)

- `IsAvailable()` can block until query is done
 - Force single-threaded mode
 - Ease debugging
- `Future<Path>` can return partial data
 - First segments of path

Asynchronous Request System (1/3)

- A lot of queries about the world are made at a constant frequency
 - Coverage value
 - Enemies' state
- If queries are multithreaded, a job should be pushed for each query executed
- The code must wait for availability of the answer

Asynchronous Request System (2/2)

- Register asynchronous request
 - At initialization of the entity :

```
Request<CoverageValue> coverage_value  
= RequestSystem::AddRequest(  
    this_entity, new CoverageRequest(...), EveryTwoFrames );
```

- During update, use Request object as a value
 - If request is not yet evaluated, it returns previous frame value
 - Don't need to write special code if data is not yet available, so your code will always work, even with outdated data

Applicability to Sensor Systems

- Easy to thread
 - No dependency with other requests
 - Don't always need to be available at a given time
- Pathfinding
 - Use job queue system
- Cover analysis, terrain strategies
 - Use asynchronous request system

Parallelism in AI

- Introduction
- Data structures
- Code and Data usage
- High-level AI architecture
- **Discussion**

Future of AI (1/2)

- « The Free Lunch Is Over »
 - Herb Sutter, Dr Dobb's Journal, March 2005
 - But languages and architectures are changing
 - Transactional memory
- Tasks are not infinitely dividable
 - Instead of trying to dispatch existing tasks on all the CPUs, use the rest of the power to increase the quality of the AI.

Future of AI (2/2)

- Animation
 - Subtle movement with influence from environment
 - Better quality of motion
 - Express emotion
- Speech synthesis and recognition
 - Better immersion of the player
- Much more entities
- ... You must contribute to this list

Performance vs Determinism

- Would you sacrifice determinism for better performances?
 - Not deterministic does not mean not predictable

References

- Herb Sutters
 - <http://www.gotw.ca/>
 - <http://herbsutter.wordpress.com/>
- AI Gamedev.com
- SPU programming, Insomniac Games, GDC08

Questions?

julien.hamaide@fishingcactus.com

