Speaker notes

Hello

My name is Dominik Grabiec

This talk is <TITLE>
Focusing on optimising the process around building the data - the data build system

- How many people are in Game Development?
- How many people have worked in AAA?
- How many are familiar with asynchronous programming?

TALK OVERVIEW

1. Background
   - What is data building?
   - Differences from Game Code
   - Assumptions and Concepts
2. Techniques
   - Keep Threads Busy
   - 3D Caching
   - Optimise Sorting
   - Avoid Blocking Threads
3. Questions

Speaker notes

Three sections

Background
What data building is
Differences from normal game code
Concepts used in presentation

Techniques I've used to optimise the data building system

Time for **questions** at end
Numbers at bottom of slides

ABOUT ME

- Programming in C++ since 1999
  (Professionally since 2005)
- Worked in Embedded and Application Development
- Since 2013 in AAA Game Development
- Representing myself
- Examples in this talk are recreations

Speaker notes

A quick bit about me

Programming since High School
Started C++ in 1999
Professionally since 2005

Worked in embedded medical devices, defence video software, web development

Got into programming to make games
Started AAA GameDev in 2013, doing ever since

Representing myself, not any employer

Examples and code are my recreations from my memory and knowledge.

BACKGROUND

WHAT IS DATA BUILDING?

- Called: compiling, converting, generating, “cooking”, or “baking”.
- Pre-computes data that is loaded by the game
  - Creates additional data from existing assets
  - Optimises and compresses data to make loading faster
- Includes arranging the data within files (and on disks)

Speaker notes

So what is data building?

Can be called compiling, converting, generating - in GameDev cooking or baking
I like data building - analogous to building code

▶ Pre-computes data for the game - does this in 2 main ways

▶ Create additional data needed by the game

From existing data made in editors

- Nav meshes - Help entities move in space
- Acoustic - Propagate sound waves
- Rendering - Make game prettier
- Other - Collect game data from world

▶ Optimise and compress data

Makes data smaller, loads faster
Removes editor & debug data
Combines loose files

▶ Arrange data on disk

Pack dependent files together - loaded together - minimise seek times

Not so much any more - more common with HDD and Optical
Not needed with SSD - seek times are zero

DATA BUILDING SYSTEMS

- Spectrum of possible designs
  - No data building – use raw text, mesh, image files in game
  - Full optimisation – use optimised custom formats
- Usually somewhere in the middle
- Design choices depend on:
  - Budget / Game size & scope
  - Studio size, Team experience
  - Existing infrastructure & technology

Speaker notes

Range of designs

▶ Minimal extreme

No building - game reads loose files on disk
Original, possibly unoptimised, formats - PNG, FBX, JSON
More common with indie games

▶ Maximal extreme

Large process - requires entire server farm
Creates lots of data - even more to process
Outputs optimised binary files - loaded quickly and efficiently
Only in large studios

▶ Usually in the middle

▶ Depends on many factors

Will focus on AAA scale - closer to the maximum

DIFFERENCES FROM THE GAME

GAME

- Frame based execution
- Minimise frame time
- Systems run together
- Read and decompress
- **Smooth player experience**

DATA BUILDING

- Batch execution
- Minimise wall clock time
- Systems run in isolation
- Read/Write and compress
- **Developer iteration time**

Speaker notes

Key differences between game and data building

Game Code

Frame centric
Minimise frame time - fast frame rate
Eliminate frame time spikes - remove jitter
Lag & jitter break immersion for the player

Code which takes milliseconds - IO, decompression, web request - moved to background threads

Data building code

Run outside of the game
Needed before game can run

Process as quickly as possible
Developers need to iterate

COMMON GAME CODE GOTCHAS

Issues encountered when reusing game code for data building code

- Writing non-thread-safe functions
- Writing singleton classes and systems
- Assuming frames are being processed
- Assuming game systems are running
- Putting data into global shared state

Speaker notes

Common game code gotchas that are encountered
Affect data building code

Functions written in thread-unsafe manner
Using function static variables - caching results
Assuming not run in parallel code

Writing systems as singletons
Assuming only one instance is needed

Assuming frame processing is happening
Filling frame related buffers - crash when full

Assuming all game systems are initialised - and running

Putting data into global shared system - like object graph, ECS, or database
Good for the game - makes processing files individually more difficult

OPTIMISATION GOAL

Minimise the time taken to process all the game data

Steps to reduce build time:

- Only build things that have changed
- Cache data locally and globally
- Stale data can be fine
- Process everything in parallel

Speaker notes

Goal is to minimise wall clock time
Steps are similar to other build systems - with changes for game data

▶ Only build what has changed - or changed by dependencies

Keep dependency info - Filenames, timestamps, and content hashes
Software version is a dependency - so is output format
Changed timestamp means recompute hash

▶ Use caches - for everything possible

Processing caches
Used internally during data building
Store frequently read values - save on IO & processing

Output caches
Store built data - Can copy rather than rebuild
Local, Office, or Company level - balance between sharing and access time

Content hashes help find data in caches
Binary determinism also important

▶ In GameDev you can cheat - Use stale data - no need to rebuild
Existing data on disk might be ok - if missing can download latest from cache
Different people care about different data at different times

Examples include Nav Mesh - or rendering data for global illumination

▶ Process everything in parallel

ASSUMPTIONS

- Built on Job System, with Async IO
- Run on single PC with many processor cores
- Fast & sizeable memory & storage
- Using 16 Physical / 32 Logical Core Processor
- Mostly C++17 and C++20 (GameDev is stuck here)

Speaker notes

Quick detour - assumptions about data build system

Techniques for running on single computer - many threads - not about clusters

Built for async processing - job system and async IO

PC used for examples is from 4-5 years ago
16 physical cores - 32 logical cores - 64GB RAM - fast NVME

Mostly C++17 and some C++20 features
What GameDev is currently using
Not all vendors support latest C++

TERMINOLOGY

- **Job**: indivisible unit of work
- **Task**: larger process converting input to output
- **SpinLock**: lock that busy waits rather than sleep
- **FlatMap**: key-value container using sorted array for keys

Speaker notes

Job means indivisible unit of work - run on single thread
Can be part of parallel operation

Task is larger process converting inputs to an output
Consists of many jobs - on many threads

SpinLock is sync primitive - spins rather than sleeping

Flat map is associative container
Stores items in arrays - keys in sorted array - giving O(logn) access

JOB SYSTEM

- Schedules jobs on many worker threads
- Uses a Counter to synchronise between Jobs
  - Leading "wait" counter marks job as runnable when reaching 0
  - Job increments "accum" counter when scheduled
  - Job decrements "accum" counter when finished
- Inspired by Naughty Dog and CD Projekt RED job systems

Speaker notes

Jobs run on worker threads
Number worker threads limited to logical processors

Counters used to sync between jobs
Each job uses two counters
Wait counter - marks job runnable when 0
Accum counter - job increments when scheduled, decrements when finished

More than 1 job per counter - easily create job graphs

Job system used in examples - based on ND and CDPR
Described in GDC presentations & Game Engine Architecture book

PROFILING

- Many Profilers used in Game Development
  Tracy, Intel VTune, Microsoft PIX, Custom
- Need profiler with instrumentation
- Need to see whole process and all threads

Using Intel's VTune in this presentation

Speaker notes

Key tool in GameDev - custom or 3rd party - many integrated into engine
Used to measure CPU, GPU, etc

Need profiler with instrumentation
Cannot rely on sampling - everything is in jobs

Need to see all threads
Looking for gaps, long jobs, job dependencies

Using VTune - best view for data building

Presentation about Tracy profiler from CppCon 2023
An Introduction to Tracy Profiler in C++ CppCon 2023

TECHNIQUES

KEEP THREADS BUSY

Keep threads busy doing **useful** work

- Make jobs roughly the same size
  - Prevents waiting on a single long job
  - Can distribute work evenly on threads
- Split long functions into jobs

Speaker notes

Main thing - keep threads busy doing useful work

▶ Best way - make all jobs roughly same size
Prevents waiting on single long job
Can distribute work more evenly

▶ Long functions should be split into jobs
Independent sections processed in parallel
Large arrays using parallel algorithms

BUILDING LARGE WORLDS

- Create read-only world cache for fast queries
- Subdivide into regions to process as independent tasks
- Some regions will take longer to process

Speaker notes

Pre-process data to create read-only world cache
Stores static world contents - for quick access - no duplicated effort
Beware of memory cost

Need to divide the world into regions - process as separate tasks
Changing one region should not affect others

Some tasks will take longer - more data to process in region
Hope all jobs are similar in size
If not then small jobs surround the larger one

Analogy is to put different sized rocks in a jar, then fill with sand to surround the larger rocks

DEALING WITH EXPONENTIALLY LONG TASKS

- Task that takes hours instead of minutes
- Could be bad data, bad algorithm, or code bug
- One method to deal with them:
  1. Build the tasks once
  2. Upload to the cache
  3. Disable local processing
  4. Make everyone download instead

Speaker notes

Sometimes tasks can take exponentially longer
Hours compared to minutes - minutes compared to seconds

Case by case examination and solution
Could be data issue - algorithm issue - or code bug

▶ One way to deal with them - build the affected tasks once - disable local processing - download from cache only

CACHING STATIC WORLD DATA

- Problems:
  - Need to cache static world data for access using 3D coords
  - Prevent tasks from duplicating IO and processing work
- Usually use traditional spatial data structures
  - KD-tree, Octree, Quadtree
  - $\mathcal{O}(\log n)$ Lookup
  - Large volume queries require multiple traversals

Speaker notes

Building large worlds requires having quick access to static world data

Many regions of the world - many tasks for each region
Need to prevent everyone from duplicating work - and code

▶ Traditionally use one of tree structures - recursively subdivide 3d space

- K-D tree partitions by axially aligned plane
- Octree into 8 octants surrounding centre point
- Quadtree into 4 quadrants
  Still useful even though is 2d - data can be 2d in nature

These provide O(logn) access
For larger area queries - multiple traversals might be required

INTRODUCING THE GRID CACHE

- Stores static world elements for quick queries
- Consider it a sparse 3D array
- Process to create:
  - Partition 3D space into cube grid
  - Distribute items into grid cells
  - Store non-empty cells in a map

Speaker notes

Presenting GridCache - alternative data structure for fast 3D lookups

Created this when developing infrastructure to build a large city
Have seen similar things - but nothing like this

Can think of it as sparse 3D array

Partition 3d space into fixed sized grid
Put from those locations items into cells
Store non-empty cells in a map - using array index as key

Ideally to simplify math and logic - form from a big cube - centered on origin

GRID CACHE MAP CHOICE

HASHMAP

- $\mathcal{O}(1)$ Lookup
- High per-cell cost
- Good for small area queries

FLATMAP

- $\mathcal{O}(\log n)$ Lookup
- No per-cell memory cost
- Good for large area queries

Speaker notes

Choice of FlatMap or HashMap depends

HashMap

Faster O(1) lookup
Higher per-cell cost - memory or runtime - depends on hashmap design

FlatMap

Slower O(logn) lookup
No per-cell overhead
Easier to gather adjacent cells - lower_bound & upper_bound

Types of queries impact which to pick
Small area queries - fit within cell - HashMap
Large area queries - contain many cells - FlatMap

Take memory into account - depending on world and cell size - could be many cells

CELL INDEX FROM 3D COORDINATE

```cpp
struct GridDimensions
{
  uint32_t cell_count;  // Number of cells in each axis
  float grid_extent;    // Grid extent from origin
  float cell_size;      // Size of cell

  uint32_t CalculateAxisOffset(float value)
  {
    value = floor(clamp(value, -grid_extent, grid_extent));
    return static_cast<uint32_t>((value + grid_extent) / cell_size);
  }

  uint64_t CalculateCellIndex(float x, float y, float z)
  {
    uint32_t ix = CalculateAxisOffset(x);
    uint32_t iy = CalculateAxisOffset(y);
    uint32_t iz = CalculateAxisOffset(z);
    return (static_cast<uint64_t>(iz) * cell_count + iy) * cell_count + ix;
  }
};
```

Speaker notes

Code from a helper object - GridDimensions

Code to calculate cell index which contains given 3d coordinates

Normalises 3d coordinates into indexes
Combine indexes into one - like multi-dimensional array index

CELL BOUNDS FROM INDEX

```cpp
struct GridDimensions
{
  uint32_t cell_count;  // Number of cells in each axis
  float grid_extent;    // Grid extent from origin
  float cell_size;    // Size of cell

  Box CalculateCellBounds(uint64_t index)
  {
    const uint32_t ix = static_cast<uint32_t>(index % cell_count);
    index /= cell_count;
    const uint32_t iy = static_cast<uint32_t>(index % cell_count);
    index /= cell_count;
    const uint32_t iz = static_cast<uint32_t>(index % cell_count);

    Vector minimum{ix * cell_size - grid_extent, iy * cell_size - grid_extent, iz * cell_size - grid_extent};
    Vector maximum = minimum + Vector{cell_size, cell_size, cell_size};
    return Box{minimum, maximum};
  }
};
```

Speaker notes

Reverse operation - calculate bounds of cell with index

Deconstruct index into 3 - compute minimum - compute maximum

GRID CACHE CLASS

- Provides query interface
- Stores a map of GridCells
- Stores bounding boxes
- Has GridDimensions
- Also a "large item" GridCell

```
class GridCache
{
public:
  bool HasNodesInBounds(const math::Box& bounds) const;
  bool HasNodesInBoundsOfType(const math::Box& bounds,
    const NodeTypes& types) const;

  std::vector<GridNode*> GetNodesInBounds(const math::Box& bounds) const;
  std::vector<GridNode*> GetNodesInBoundsOfType(const math::Box& bounds,
    const NodeTypes& types) const;

  const GridCellPtr& GetLargeGridCell() const;
  GridCellPtr GetGridCell(const math::Vector& point) const;
  std::vector<GridCellPtr> GetGridCells(const math::Box& bounds) const;

private:
  Map<uint64_t, GridCellPtr> grid_cells_;
  GridCellPtr large_cell_;

  math::Box grid_bounds_;
  math::Box world_bounds_;
  GridDimensions grid_dimensions_;
};
```

Speaker notes

GridCache class - root interface object - has query functions

Stores map of GridCells - bounds of grid and world - GridDimensions object

Need to have "large item" GridCell - quarantine items with massive bounds
Example of light covering the world

GRID CELL CLASS

- Array of "owned" GridNodes
- Array of "intersecting" GridNodes
- Bounding box of cell

```cpp
class GridCell
{
public:
  bool HasNodesOfType(const NodeTypes& types) const;

  std::vector<GridNode*> GetNodesOfType(const NodeTypes& types) const;
  std::vector<GridNode*> GetAllNodes() const;

  void AppendNodesOfType(const NodeTypes& types,
    std::vector<GridNode*>& result) const;
  void AppendAllNodes(std::vector<GridNode*>& result) const;

private:
  Box bounds_;

  std::vector<GridNode*> owned_nodes_;
  std::vector<GridNode*> intersecting_nodes_;
};
```

Speaker notes

GridCell stores sorted arrays of GridNodes

- Array of owned nodes - cell contains bounding box origin
- Array of intersecting nodes - cell intersects bounding box

Different arrays are useful for different queries
Items which belong to a region - use owned nodes
Items which influence / intersect a region - use intersecting nodes

GRID NODE CLASS

- Actually stores the data
- Has node type
- Has bounding box & world transform
- Allocated in large buffer
- Keep small and tightly packed
- Referenced by raw pointers

```
struct alignas(32) GridNode
{
  NodeType type;              // 4 bytes
  uint32_t properties_hash;   // 4 bytes
  ResourcePath path;          // 8 bytes
  math::Box bounds;           // 32 bytes
  math::Transform transform;  // 32 bytes
  math::Vector scale;         // 16 bytes
  GlobalId global_id;         // 8 bytes
  GameDataPtr game_data;      // 8 bytes
  GridNode* parent;           // 8 bytes
  uint32_t sort_order;        // 4 bytes
  uint32_t flags;             // 4 bytes
};
static_assert(sizeof(GridNode) == 128);
```

Speaker notes

GridNodes store data to be cached - including positional

Node type - resource path - bounding box - transform
Also stores other values - ones here are examples

Store what is needed - but no more - likely to be tens of millions
Every byte counts - check alignment and padding
Use bitfields & other tricks

Allocated in external array - referenced by raw pointer

Sorting needs to be quick - done many times
Both at construction and during queries

SORTING GRID NODES

```
inline bool operator<(const GridNode& left, const GridNode& right)
{
  return std::tie(left.type, left.properties_hash, left.path, /*...*/)
    < std::tie(right.type, right.properties_hash, right.path, /*...*/);
}
```

```
inline bool operator<(const GridNode& left, const GridNode& right)
{
  if (left.type == right.type)
  {
    if (left.path == right.path)
    {
      if (left.properties_hash == right.properties_hash)
      {
        // ...
      }
      return left.properties_hash < right.properties_hash;
    }
    return left.path < right.path;
  }
  return left.type < right.type;
}
```

```
inline bool operator<(const GridNode& left, const GridNode& right)
{
  if (left.type == right.type)
  {
    if (left.path == right.path)
    {
      if (left.properties_hash == right.properties_hash)
      {
        return GridNodeInternalLess(left, right);
```

```
    }
    return left.properties_hash < right.properties_hash;
  }
  return left.path < right.path;
}
return left.type < right.type;
}
```

```
bool GridNodeInternalLess(const GridNode& left, const GridNode& right)
{
  if (left.bounds == right.bounds)
  {
    if (left.transform == right.transform)
    {
      if (left.scale == right.scale)
      {
        //...
        return left.sort_order < right.sort_order;
        //...
      }
      return left.scale < right.scale;
    }
    return left.transform < right.transform;
  }
  return left.bounds < right.bounds;
}
```

Speaker notes

Brings us to optimising sorting of GridNodes - optimising less than operator

▶ Quickest to write is using std::tie - create tuples & compare
Negatively impacts compile performance - each file parses & evaluates tuple templates
Also impacts runtime performance - in debug mostly - also in release

▶ Easiest fix - expand code manually
Fixes compile time issue - no templates
Function too big to be inlined by compiler - just like std::tie release

▶ Better fix - split code into fast inline - slow non-inline functions
Put important comparisons inline - put remainder in source file
Goal to return from inline code in nearly all cases

First sort by how to organise - by type to make queries easier
This example - sort by type first - then by path - then by properties hash
If objects are equal we compare more but slower

▶ Include tie-breaker value - called sort-order - to distinguish duplicate nodes
Index of when node was loaded

OPTIMISING SORTING OF LARGE ARRAYS

- Sorting large arrays can be expensive
- Sorting algorithms are $\mathcal{O}(n \log n)$
- Expensive comparison functions hurt performance
- Elements not used in sorting are a waste of cache
- Optimising for memory and cache

Speaker notes

Sorting large arrays can be expensive - even with parallel sort

Main reason - O(nlogn) complexity
Comparison function evaluated n.logn times
Memory evaluated logn times

Large items waste cache
Slow compare function - wastes cpu - indirection is expensive

Sorting is generally fast - but optimising can save a second or two

ORIGINAL NODE SORTING CODE

```cpp
struct Node;
using NodePtr = std::shared_ptr<Node>;

struct Prefab
{
  uint64_t GetId() const;

  std::vector<NodePtr> nodes_;
};

struct Node
{
  virtual ~Node();
  Prefab* GetOwner() const;
  uint64_t GetId() const;
};

struct PrefabNode : public Node;
struct MeshNode : public Node;
struct EntityNode : public Node;
```

```cpp
bool SortFunc(const NodePtr& left, const NodePtr& right)
{
  auto left_prefab_id = left->GetOwner()->GetId();
  auto right_prefab_id = right->GetOwner()->GetId();
  if (left_prefab_id == right_prefab_id)
  {
    return left->GetId() < right->GetId();
  }
  return left_prefab_id < right_prefab_id;
}

void SortNodes(std::vector<NodePtr>& nodes)
{
  ParallelSort(nodes.begin(), nodes.end(), SortFunc);
}
```

Speaker notes

Shows original code - typical example of prefab and node in 3D world
Stripped down example - real life these contain many functions

Comparison is expensive - because of indirection

Sorting tens of million of nodes takes time

HOW TO OPTIMISE DATA FOR SORTING

1. Extract needed values into separate sorting array
   With index of original location
2. Sort the smaller sorting array
3. Create result array
   Move items using index from sorting array
4. Return the new result array

Also works for sorting large items

Speaker notes

Create smaller sorting struct - with variables to sort by - and original index

Extract needed values into another struct

Sort values in smaller struct

Create result array - move values in from original

Swap or move result to original

OPTIMISED NODE SORTING CODE

```
struct NodeSortData
{
  uint64_t prefab_id;
  uint64_t node_id;
  size_t original_index;

  static void Fill(NodeSortData& sort_item, const NodePtr& node, size_t index);
  static bool Compare(const NodeSortData& left, const NodeSortData& right);
};

void NodeSortData::Fill(NodeSortData& sort_item, const NodePtr& node, size_t index)
{
  sort_item.prefab_id = node->GetOwner()->GetId();
  sort_item.node_id = node->GetId();
  sort_item.original_index = index;
}

bool NodeSortData::Compare(const NodeSortData& left, const NodeSortData& right)
{
  if (left.prefab_id == right.prefab_id)
  {
    return left.node_id < right.node_id;
  }
  return left.prefab_id < right.prefab_id;
}
```

Speaker notes

Small sort struct - contains only needed elements
Also two helper functions

Fill function want to fill in parallel
Minimal comparison function

OPTIMISED NODE SORTING CODE

```
void SortNodes(std::vector<NodePtr>& nodes)
{
  std::vector<NodeSortData> sorting_data{nodes.size(), {}};
  DispatchParallelJob(nodes.size(), [&](size_t index)
  {
    NodeSortData::Fill(sorting_data[index], nodes[index], index);
  });

  ParallelSort(sorting_data.begin(), sorting_data.end(), NodeSortData::Compare);

  std::vector<NodePtr> result(nodes.size(), {});
  DispatchParallelJob(nodes.size(), [&](size_t index)
  {
    result[index] = std::move(nodes[sorting_data[index].original_index]);
  });

  std::swap(nodes, result);
}
```

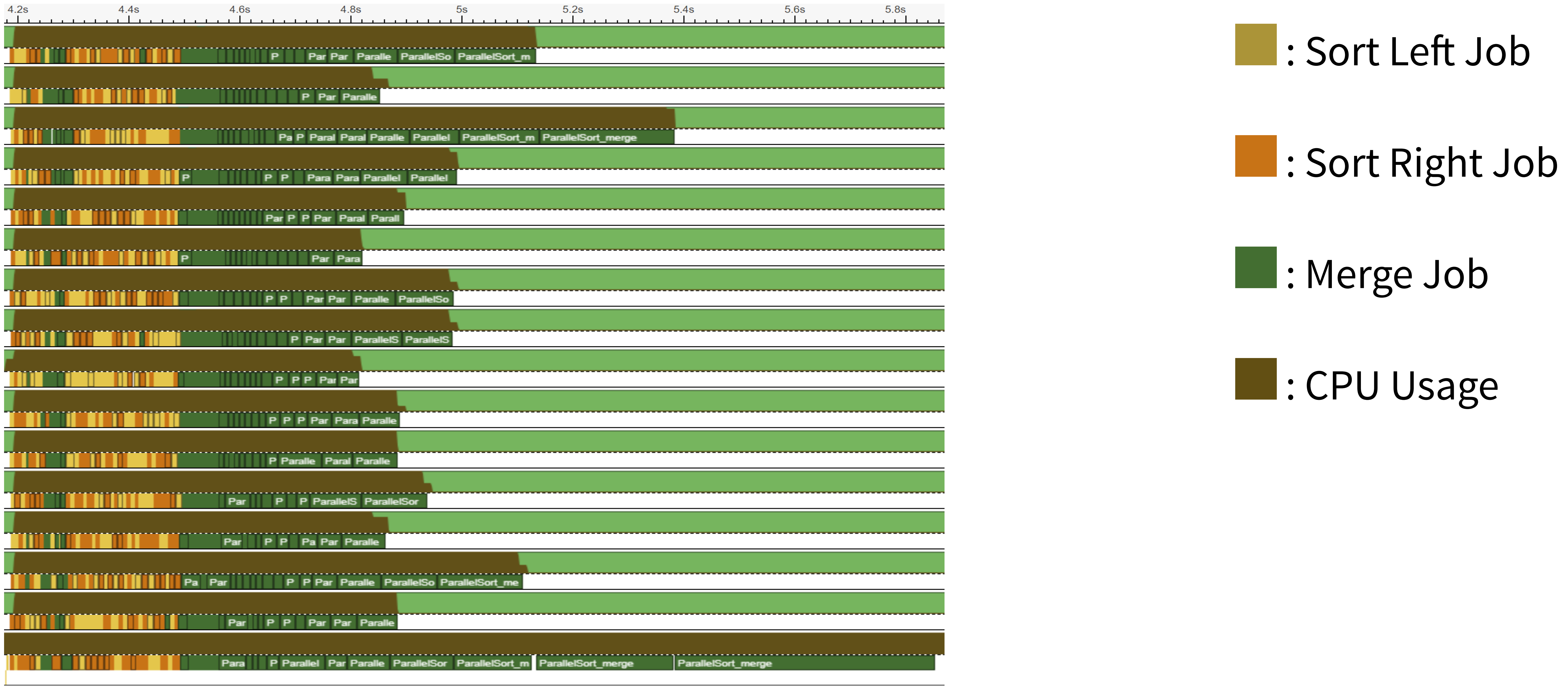
Speaker notes

Even though this is more code - only processes the full array twice rather than logn times

Sorts a much smaller array - much simpler comparison function

- Line 3 - create sorting array
- Lines 4-7 - dispatch parallel jobs to fill the array
- Line 9 - do the sort
- Line 11 - create result array
- Lines 12-15 - dispatch parallel jobs to move from original to result array
- Line 17 - swap to return the result array

ORIGINAL PROFILER CAPTURE

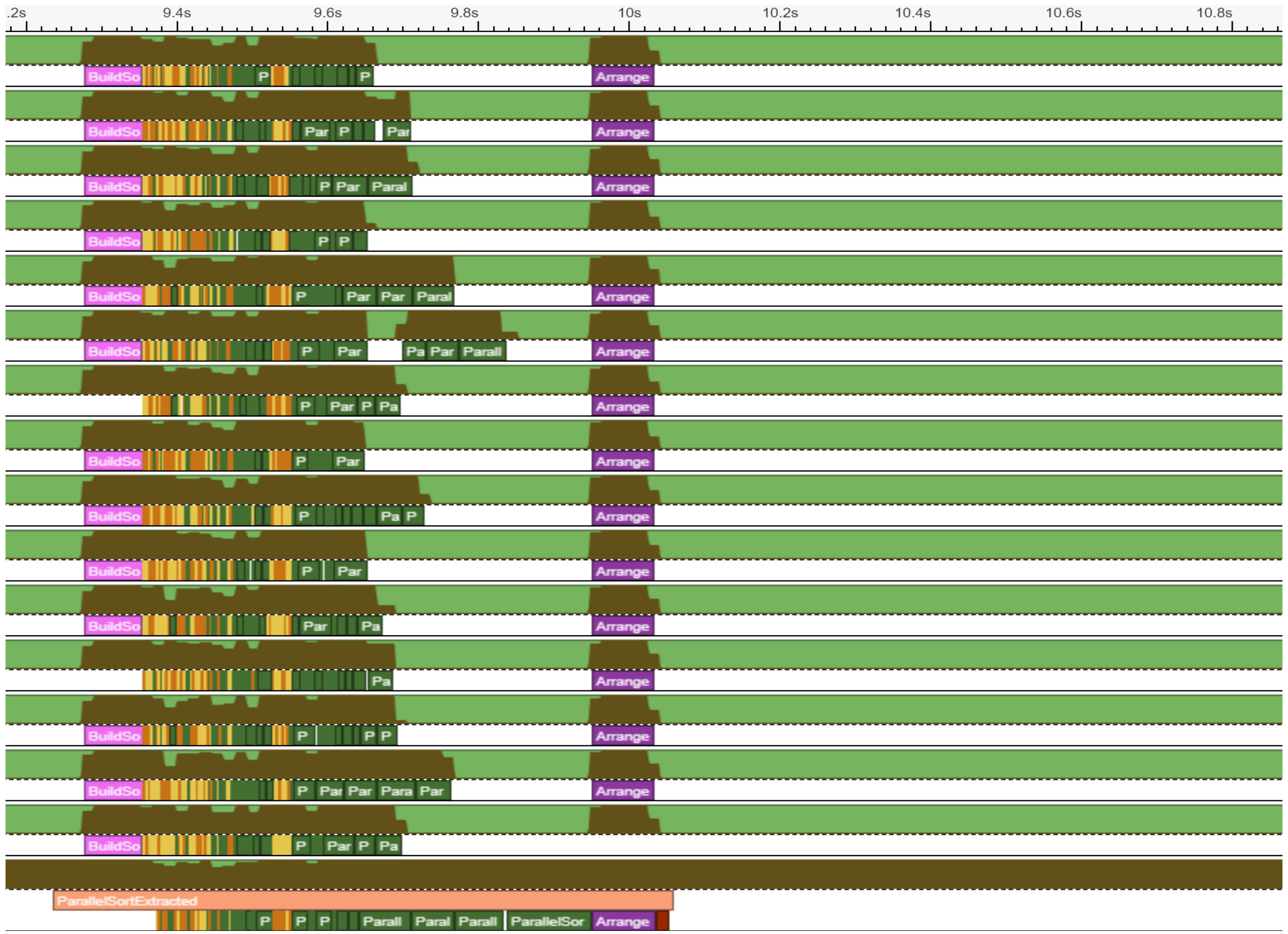

4.2s
4.4s
4.6s
4.8s
5s
5.2s
5.4s
5.6s
5.8s
: Sort Left Job
: Sort Right Job
: Merge Job
: CPU Usage

Speaker notes

Profiler capture of parallel sort of original data

Note merge results jobs at bottom

OPTIMISED PROFILER CAPTURE



: Sort Left Job

: Sort Right Job

: Merge Job

: Fill Job

: Assign Job

: CPU Usage

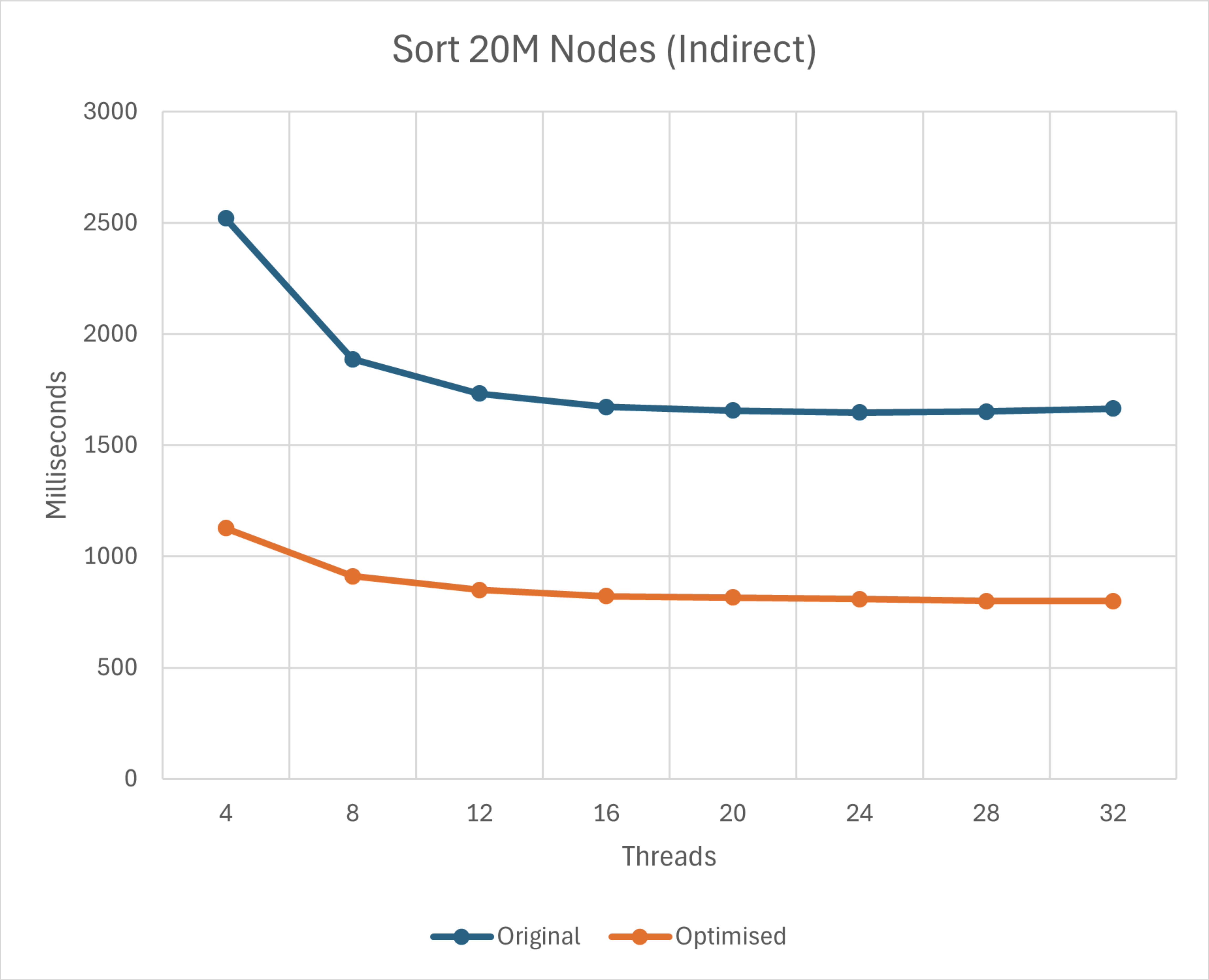
Speaker notes

Profiler capture of parallel sort of minimised data

Scale is the same as previous - takes lot less time than original

Code looks like its doing more work - but actually doing less work

OPTIMISED SORTING RESULTS



Speaker notes

Shows results when run on different thread counts

Tapering off happens around 16 threads which is physical processor cores
Could also be limitation of my test code

Also got similar results when applying this to sorting large nodes

AVOID BLOCKING THREADS

- Locking can block an entire worker thread
- Prevents other jobs from running
- Sometimes it is unavoidable
- Be aware of hidden locks
- Use the job system to synchronise instead

Speaker notes

Next optimisation technique
Avoid blocking with thread-centric sync in jobs
Mutex, SpinLock, blocking IO

Worst thing to wait on lock
Prevents other jobs from running

Best case will be no contention and no effect - worst case blocks all worker threads
Gets worse with more threads

▶ Some locks unavoidable - present in key systems - or in libraries
Sometimes in places you don't expect - logging, printf, locale

Ideally no contention on these locks - so threads aren't blocked for long time

▶ In other cases - use job system to synchronise access instead

AVOID BLOCKING – PROBLEM

- Heavy processing code
- Needed to serialise files sequentially
- SpinLock was go-to synchronisation primitive
- Result: 100% CPU usage for many minutes

```cpp
SpinLock data_save_lock;

void Process(std::shared_ptr<Data> data)
{
  data->Processing();

  {
    SpinLockGuard<SpinLock> guard(data_save_lock);
    data->SaveFile();
  }
}
```

Speaker notes

Example of this happening

Background

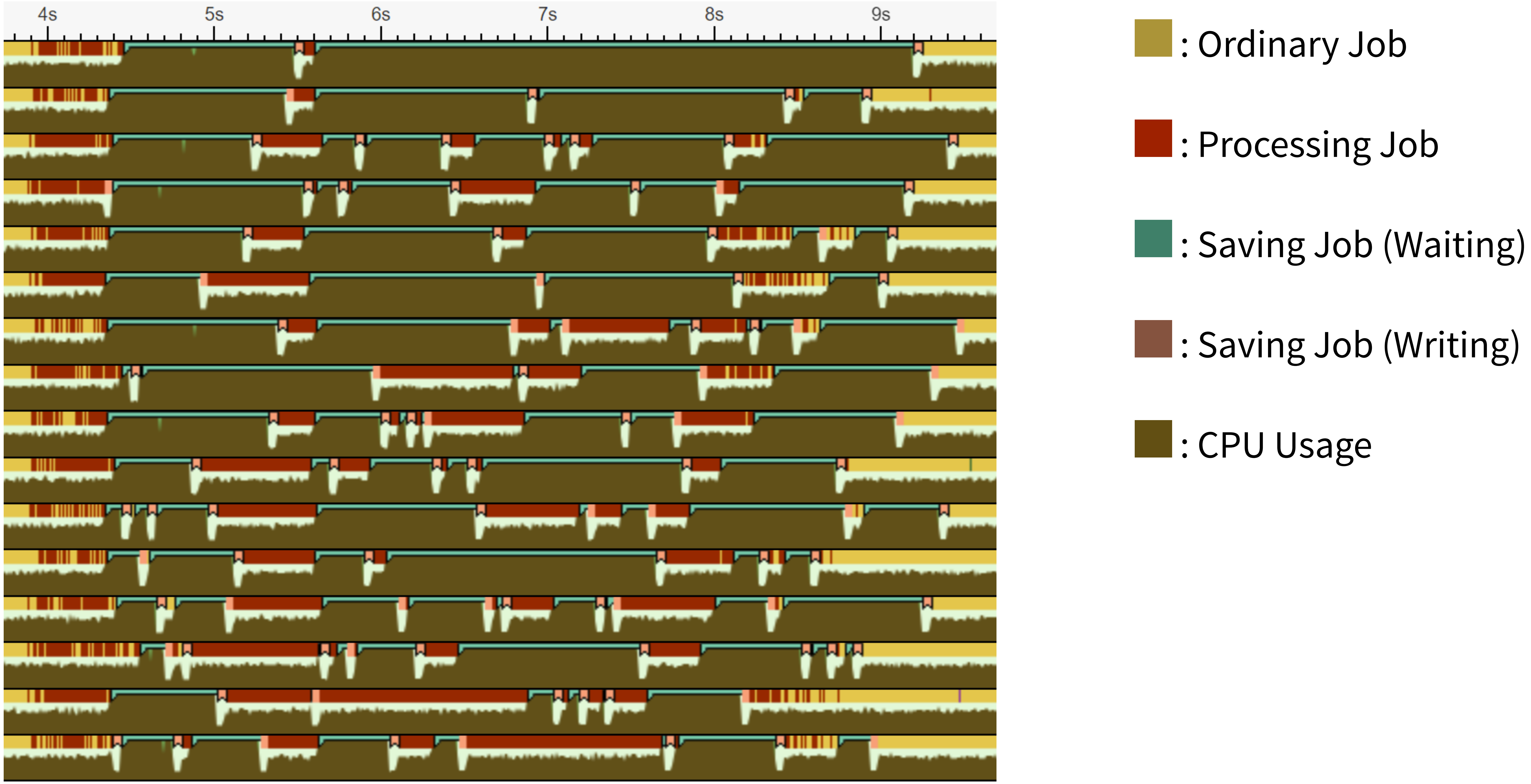
System processing lots of files - lots of shared data
Processing could happen in parallel - saving to disk could not
Using SpinLock for sync - is default for engine code - doesn't do system call

▶ **Result**

System was running 100% CPU for many minutes
Greeted by complaints in morning
People couldn't multitask while building data

Ran in profiler - found offending SpinLock & code

Single lock for 20 threads - odds are thread is spinning - rather than processing
Not likely to get lock either - it is 1 of 19 waiting

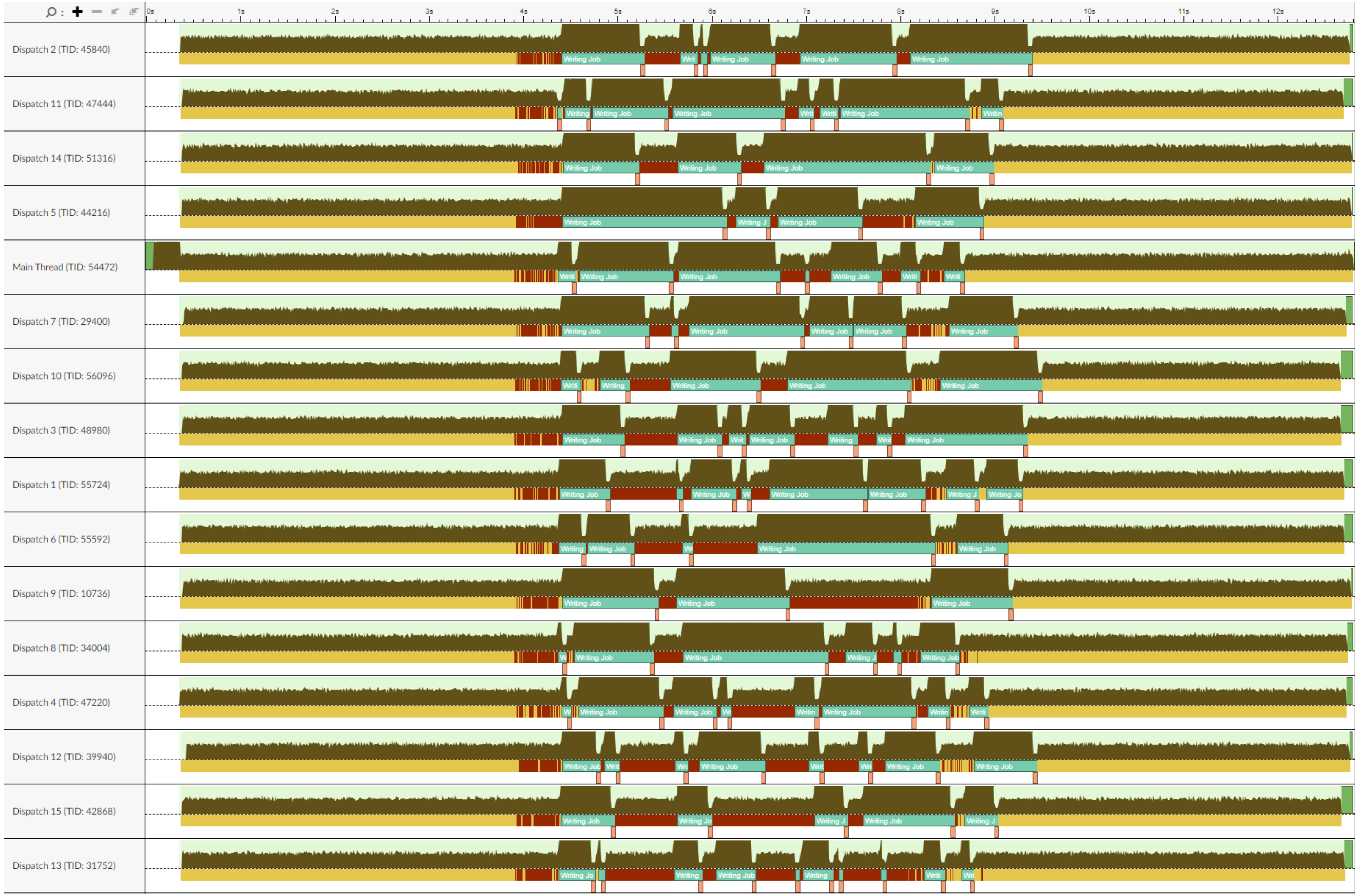

4s
5s
6s
7s
8s
9s
: Ordinary Job
: Processing Job
: Saving Job (Waiting)
: Saving Job (Writing)
: CPU Usage

Speaker notes

Profiler captures of simulated example

- Yellow - ordinary jobs - capped at 50% CPU
- Red - ordinary jobs part of saving - capped at 50%
- Teal - saving jobs

Easier to see here - I've made regular jobs use 50% CPU
100% usage is SpinLocks - Followed by 20% usage in saving jobs



Speaker notes

AVOID BLOCKING – CONFIRMATION

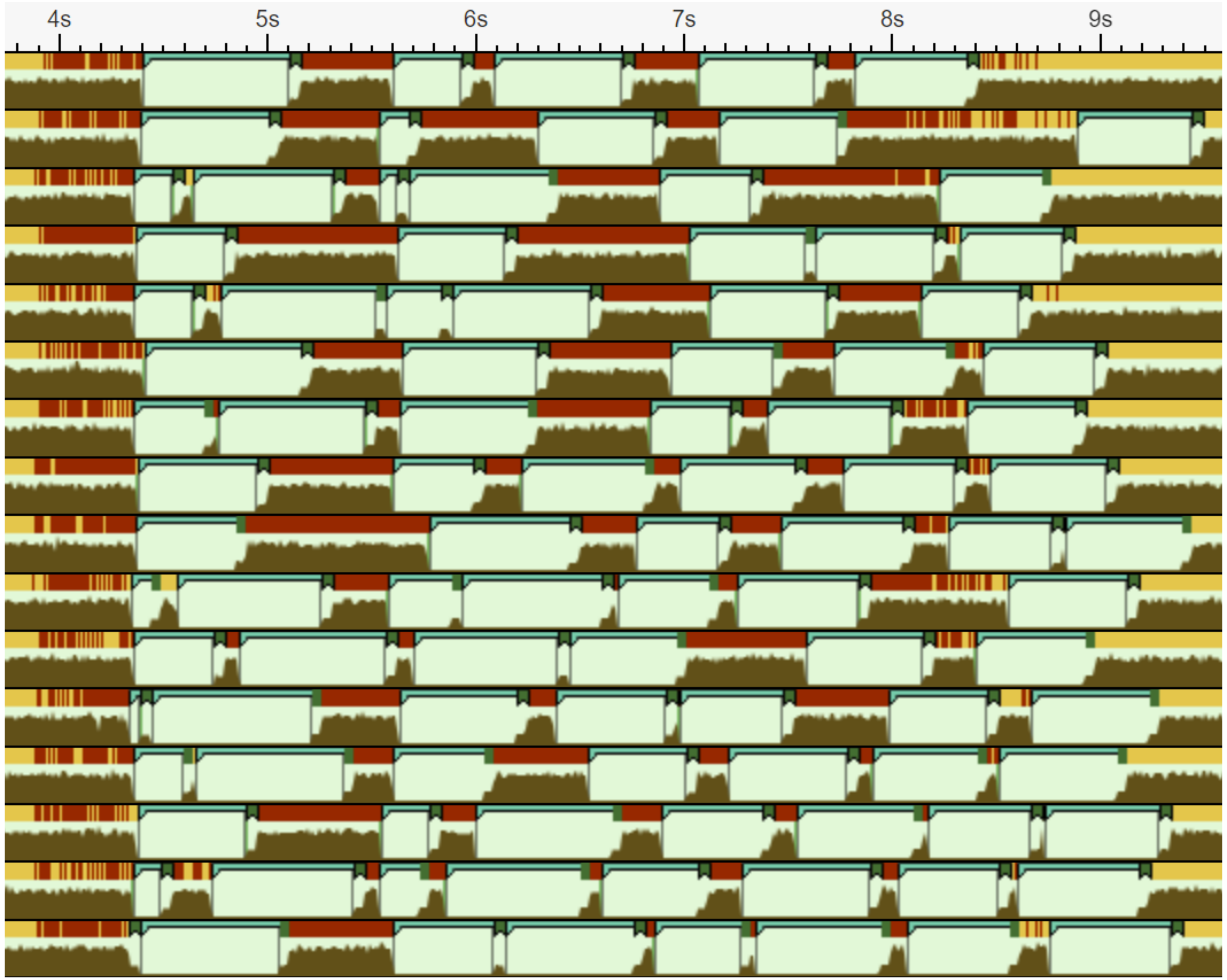
- Changed from SpinLock to mutex
- Result:
  - CPU usage dropped to single thread level
  - Confirmed that it was lock contention
  - But time taken remained the same

```cpp
std::mutex data_save_lock;

void Process(std::shared_ptr<Data> data)
{
  data->Processing();

  {
    std::lock_guard<std::mutex> guard(data_save_lock);
    data->SaveFile();
  }
}
```

Speaker notes

To verify - change from SpinLock to mutex

▶ Time taken the same - CPU usage down to minimal



: Ordinary Job

: Processing Job

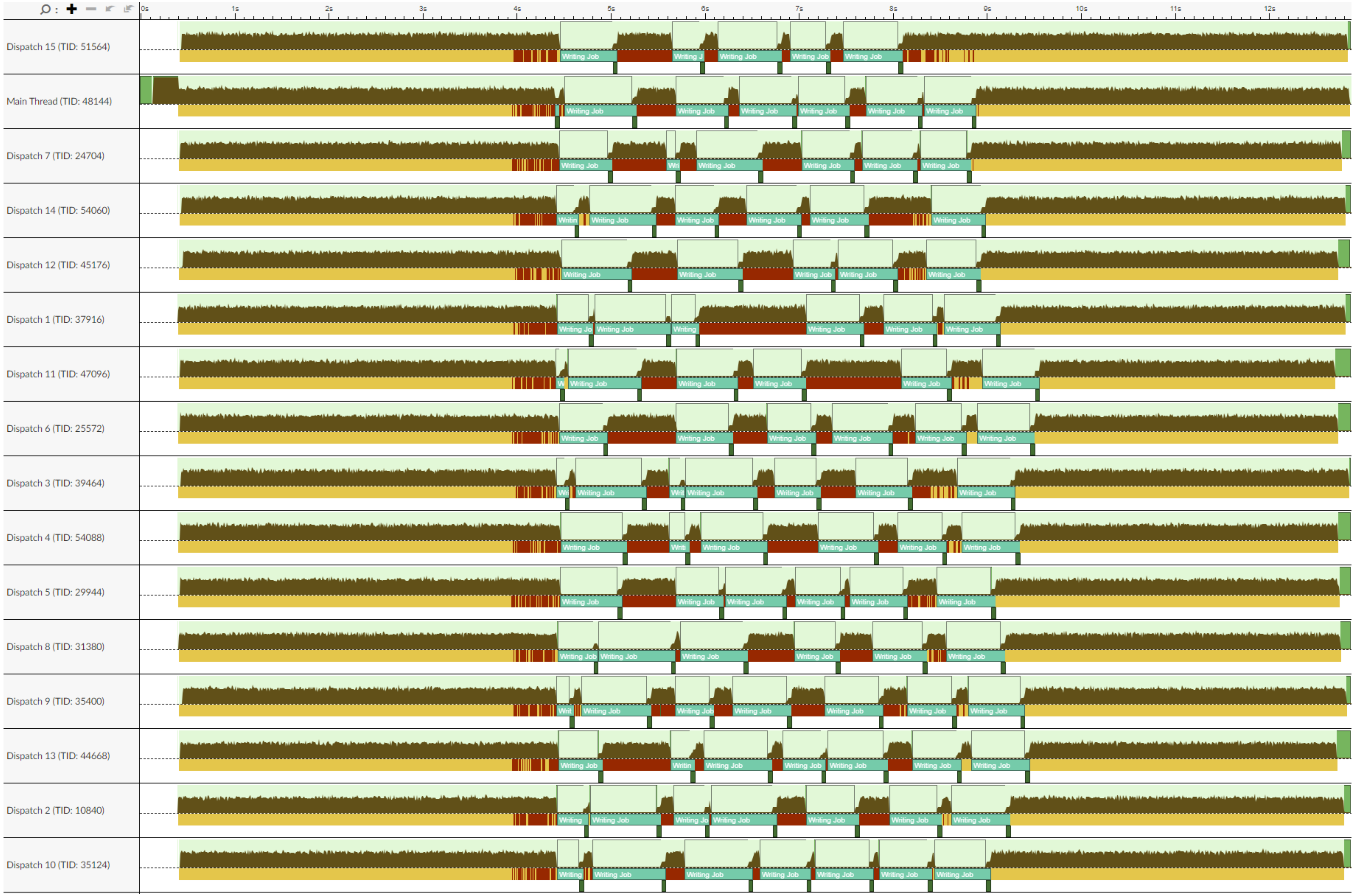
: Saving Job (Waiting)

: Saving Job (Writing)

: CPU Usage

Speaker notes

Same as before - except 100% usage replaced by 0%


0s
1s
2s
3s
4s
5s
6s
7s
8s
9s
10s
11s
12s
Dispatch 15 (TID: 51564)
Main Thread (TID: 48144)
Dispatch 7 (TID: 24704)
Dispatch 14 (TID: 54060)
Dispatch 12 (TID: 45176)
Dispatch 1 (TID: 37916)
Dispatch 11 (TID: 47096)
Dispatch 6 (TID: 25572)
Dispatch 3 (TID: 39464)
Dispatch 4 (TID: 54088)
Dispatch 5 (TID: 29944)
Dispatch 8 (TID: 31380)
Dispatch 9 (TID: 35400)
Dispatch 13 (TID: 44668)
Dispatch 2 (TID: 10840)
Dispatch 10 (TID: 35124)
Writing Job

Speaker notes

AVOID BLOCKING – SOLUTION

- Refactored function to use more jobs
- Isolated saving code in own job
- Created second dependency chain for saving jobs
- Allowed other tasks to run in parallel while saving was serial
- Result: Was able to do more useful work and faster

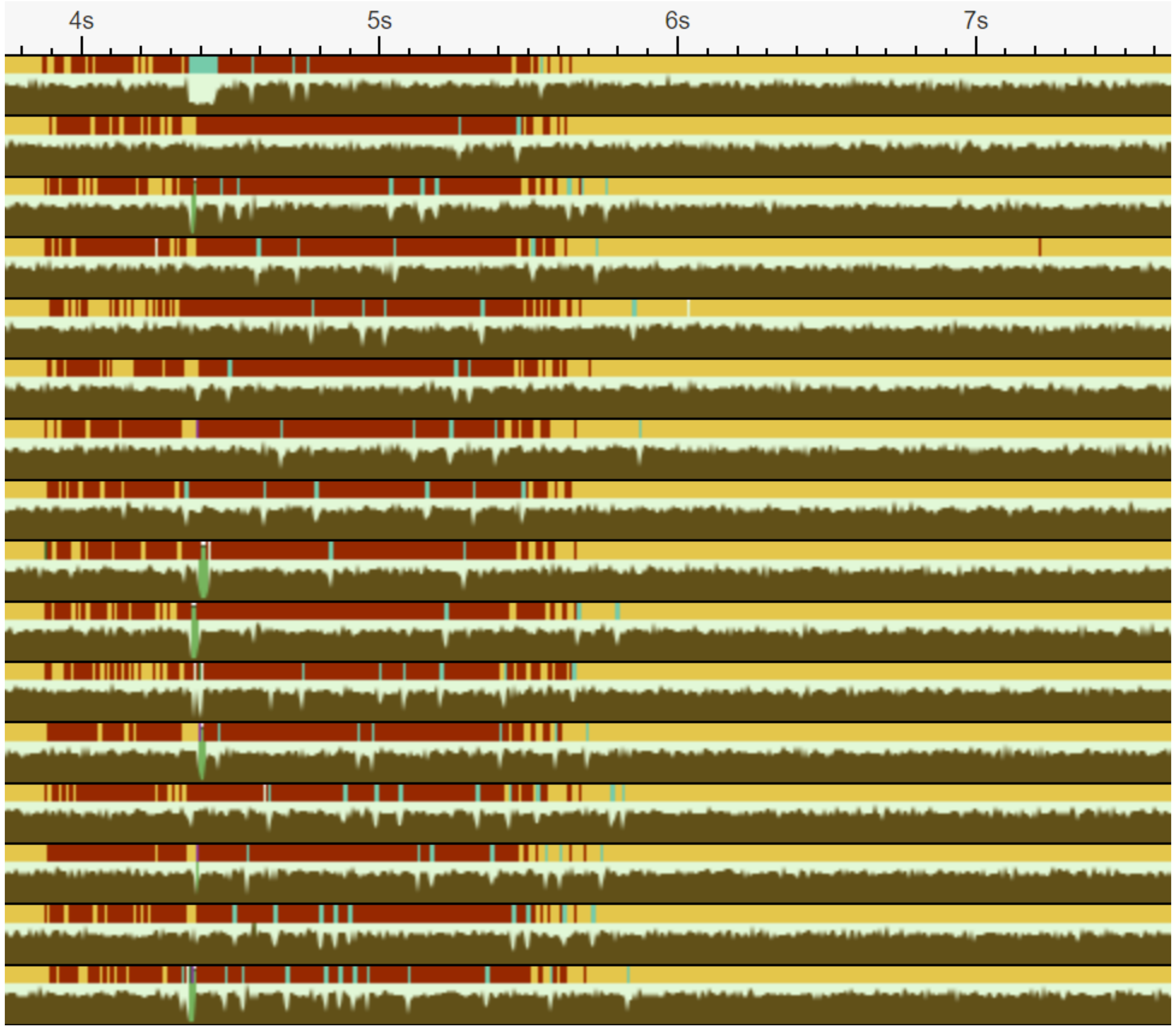
```
std::mutex data_save_lock;
job::Counter data_save_counter;

void Process(std::shared_ptr<Data> data)
{
  DispatchJob(start_counter, save_counter,
    [data](){ data->Processing(); });

  {
    std::lock_guard<std::mutex> guard(data_save_lock);
    save_counter += data_save_counter;
    DispatchJob(save_counter, post_counter,
      [data]() { data->SaveFile(); });
    data_save_counter = post_counter;
  }
}
```

Speaker notes

Thought about the problem - solution was to use job system - rather than thread sync primitive

First step - refactor code to use more jobs
Put saving code into own job
Add new dependency chain for these jobs
Effectively create a linked list of saving jobs

▶ Did not reduce time to save data - Did allow other jobs to run
Significant reduction in total time



: Ordinary Job

: Processing Job

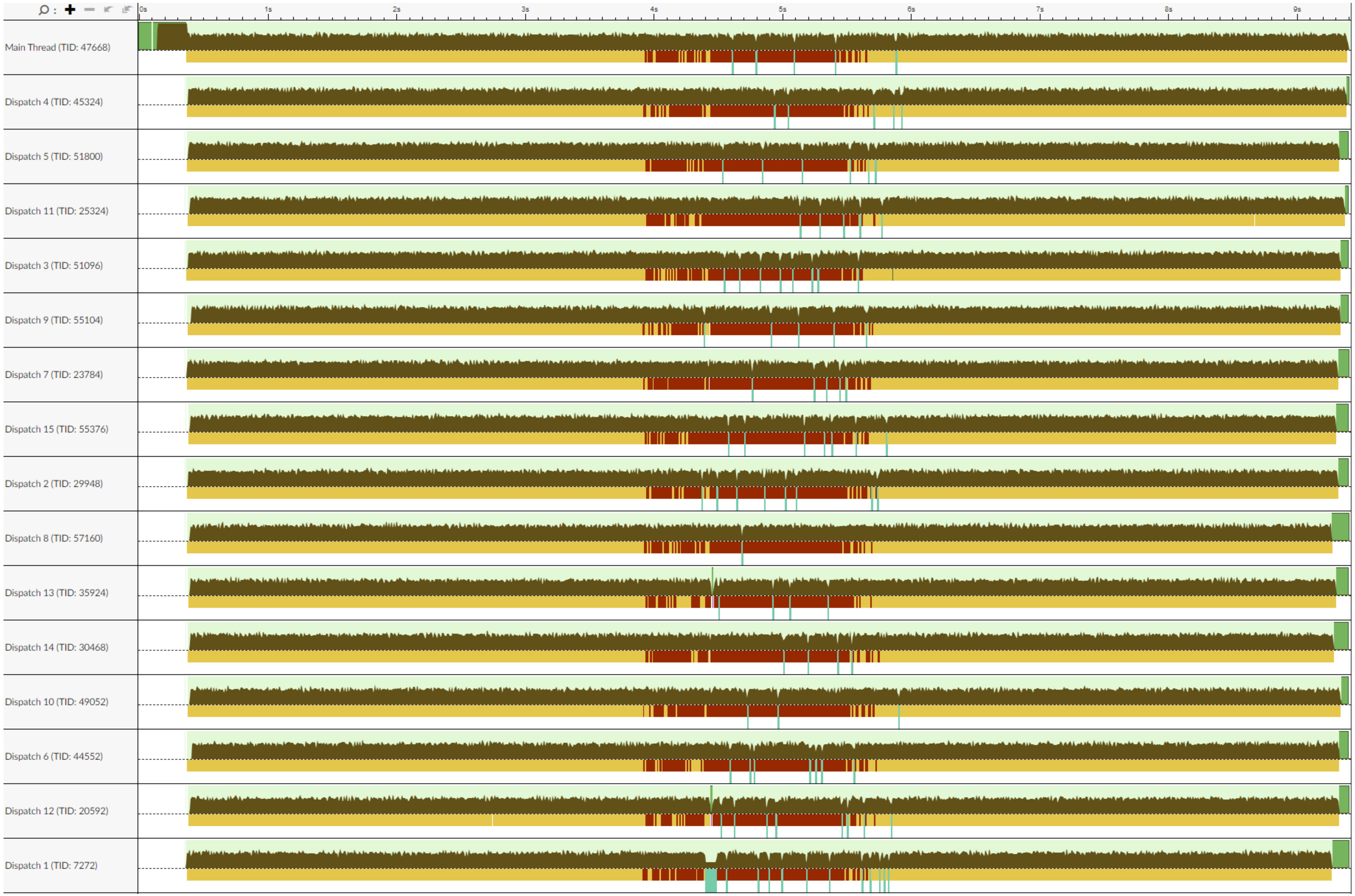
: Saving Job

: CPU Usage

Speaker notes

Can barely see any teal saving jobs

Total time reduced by over 2 seconds - in all cases
Significant % difference



0s
1s
2s
3s
4s
5s
6s
7s
8s
9s
Main Thread (TID: 47668)
Dispatch 4 (TID: 45324)
Dispatch 5 (TID: 51800)
Dispatch 11 (TID: 25324)
Dispatch 3 (TID: 51096)
Dispatch 9 (TID: 55104)
Dispatch 7 (TID: 23784)
Dispatch 15 (TID: 55376)
Dispatch 2 (TID: 29948)
Dispatch 8 (TID: 57160)
Dispatch 13 (TID: 35924)
Dispatch 14 (TID: 30468)
Dispatch 10 (TID: 49052)
Dispatch 6 (TID: 44552)
Dispatch 12 (TID: 20592)
Dispatch 1 (TID: 7272)

Speaker notes

AVOID LOCKING – USE JOB SYSTEM

ORIGINAL

```cpp
std::mutex data_save_lock;

void Process(std::shared_ptr<Data> data)
{
  data->Processing();

  {
    std::lock_guard<std::mutex> guard(data_save_lock);
    data->SaveFile();
  }
}
```

IMPROVED

```cpp
std::mutex data_save_lock;
job::Counter data_save_counter;

void Process(std::shared_ptr<Data> data)
{
  DispatchJob(start_counter, save_counter,
    [data](){ data->Processing(); });

  {
    std::lock_guard<std::mutex> guard(data_save_lock);
    save_counter += data_save_counter;
    DispatchJob(save_counter, post_counter,
      [data]() { data->SaveFile(); });
    data_save_counter = post_counter;
  }
}
```

Speaker notes

Original Code

1. Global mutex
2. Some processing
3. Lock and save

Improved Code

1. Global counter & lock for it - line 1 & 2
2. Dispatch job to do processing
3. Lock the mutex for the read modify write of the counter
4. Add dependency on the global counter to the local job
5. Dispatch save job
6. Replace global counter with job's counter

Creates chain of dependencies between jobs
Can think of global counter as storing pointer to the last link

SUMMARY

- Process in parallel - by making jobs roughly the same size
- Cache data when building worlds
- Optimise sorting - by minimising data and simplifying comparisons
- Avoid blocking threads - use the job system intelligently

Speaker notes

Quick summary of the techniques

ACKNOWLEDGEMENTS

- My Wife, Anna
- "Nowogrodzka" Crew (from CD Projekt RED)
  - Charles Tremblay
  - David Block
  - Tommi Nykopp
  - Wil McVicar
- Technology Team at Techland

Speaker notes

Friends and former colleagues from CD Projekt RED

Colleagues from the Technology team at Techland

For helping practice and refine this presentation

CONTACT

- Blog – https://www.dominikgrabiec.com
- GitHub – https://github.com/DominikGrabiec
- Mastodon – dominikg@gamedev.place
- Twitter - @Daemin
- LinkedIn - https://www.linkedin.com/in/dominik-grabiec-4866288

QUESTIONS

REFERENCES

- Job System
  - GDCVault - Parallelizing the Naughty Dog Engine
  - GDCVault - The Job System in Cyberpunk (Slides)
  - Game Engine Architecture, 3rd Edition Chapter 8.6 Multiprocessor Game Loops