Re-making the Movie-making Machine*

James Vanns[1] and David Galeano[1]

Industrial Light & Magic, a Lucasfilm Ltd. company.
jvanns@ilm.com
https://www.ilm.com

Abstract. The visual effects (VFX) industry has for decades been actively developing and running job and resource management systems (JARMS) under the guise of "render farms". Rarely have we sought to share our work, stories, and problems beyond our niche bubble with the wider HPC community. With this paper we hope to break that cycle and introduce you to how we at Industrial Light & Magic (ILM) support our ever-changing diverse workloads at scale to produce high quality imagery for major motion pictures and television such as *Jurassic World* and *The Mandalorian*. We then share recent development efforts to prepare us for our biggest challenge yet: *ABBA Voyage*. These developments are the result of practical improvements to a production scheduler yielding higher throughput, reduced waste and makespan, and increased insight into internal metrics. We also briefly discuss how we developed tooling to aid in trace-based simulations to gain confidence in production upgrades.

Keywords: Scheduling · Render farm · Batch schedulers · High Performance Computing · Cluster Management

1 Introduction

The VFX industry has relied upon compute farms to parallelise the tasks required to render frames for film since the mid-to-late 1990s. For ILM this has scaled from a few dozen MIPS cores at a single location to nearer 100k x86 cores per location, now numbering 5 across the globe.

A VFX shot (generally a run of frames uninterrupted by a cut or edit) may comprise of, but is not limited to, camera tracking, environment or set extension, 3D modelling, animation, texture painting, FX & Simulations, lighting & shading, and 2D compositing. These disciplines work on a single shot as it moves towards completion, and the process is colloquially referred to as "the pipeline" (although this is a simplification). Many of these stages submit work as jobs to the farm and range in complexity depending on the shot.

As an artist iterates over refinements to their work, many versions are "rendered" on the farm. For each re-render, adjustments to the resource requirements plus

* Supported by Industrial Light & Magic.

parameters specific to the DCC (digital content creation) tool are made, though they are opaque to the scheduler.

With such a heterogeneous workload running on the farm, where task types are diverse, the corresponding resource requirements are also varied. The precedent constraints between tasks that model the job structure also change with this diversity, contributing to the overall system complexity.

1.1 Basic Concepts

For ILM, concurrency comes in many forms; we work on multiple shows at a time (e.g. *The Book of Boba Fett*, *Eternals*, *Red Notice* etc.) and these are often at different stages of development and vary in complexity. A single shot can be developed by different departments at the same time (e.g. modelling, animation) especially when these iterations are versioned. Finally, we achieve task concurrency on the farm by executing many tasks simultaneously through parallel directed acyclic graphs (DAGs).

Shows generally work to different deadlines (internal, director approval, trailers etc.) and many shots are prioritised over others according to these deadlines (due dates). Individual users themselves may also wish to rank their own submissions to the farm. However, the farm is a company resource and digital resource managers (DRMs) have the tricky job of juggling conflicting objectives such as farm throughput/high utilisation vs. competing show priorities & resource contention.

Finally, each discipline or stage in this VFX pipeline demands diverse resources of varying quantities to accomplish their task. Consumable resources available on a host include CPU, GPU, RAM and local disk space (we don't factor in network bandwidth). Static resources include things such as OS release, GPU chipset, SIMD instruction set etc.

From a global pool, consumable tokens include software licences, often expensive resources hotly contended between shows. Each vendor licence may come with a different sharing scheme - an opportunity to pack and cut costs.

All these packing dimensions lead to task shapes and corresponding holes on compute nodes ranging from 1 CPU and 512MB RAM to 96 CPUs and 128GB RAM, and every combination & permutation in between. These shapes can of course influence the runtime of the task, assuming the task can parallelise well (many renderers do) on a multi-core host. Task types are so diverse that normalised runtimes range from 30 seconds (e.g data moves) to several days (e.g. machine learning) and are bounded by different resources (e.g. I/O or GPU).

It's worth noting that across VFX, rarely is a runtime or duration estimate (from the user) assigned to a task and doesn't contribute to scheduling decisions made by the system. Historically, it's been widely accepted that workloads are simply too unpredictable since many commands, once they execute, are black boxes to

the scheduler; scene parameters in the input file(s) significantly affect the render times and resources consumed.

2 Background

Star Wars Episode I: The Phantom Menace began production in 1997 and since then ILM adopted use of it's proprietary ObaQ/DOALL [6] distributed scheduling system. This system delivered hundreds of feature films and millions of frames until it was eventually superseded 2 decades later in 2020 by a centralised dispatch system, Coda [11], developed at a sibling company, Walt Disney Animation Studios (WDAS).

2.1 Coda

In 2017 work began to adapt the WDAS system for the ILM pipeline. During this period we developed additional Coda tooling, maintained & supported the existing ObaQ system, integrated Coda into our software stack all the while still delivering films. By 2019 we were coordinating the production of rendered frames through Coda and had scheduled ObaQ for decommission. As ILM began adopting version 5.1 of Coda, WDAS continued to develop version 6. We were unable to interupt our own integration and migration plans to accommodate this additional change. Therefore, ILM remained on the 5.1 release, which is the codebase we forked and maintained internally.

2.2 Production Proven

Coda 5.1 had already proven itself in production both at WDAS and ILM. At Disney, Coda had delivered animation features such as *Frozen* and *Moana* and at ILM, before the advances in this paper were required, we delivered, among others, *Star Wars: The Rise of Skywalker*. The success of Coda at both studios, despite their workflows being different, gave us confidence enough to develop it further to meet the demands of more challenging shows.

2.3 Architecture

The Coda architecture is designed such that system responsibilities reflect components often shared by similar systems (see Fig. 1). The reflector is a central communications broker where every message flows through it via topical channels - every other Coda component communicates through it; there is no other direct peer transport. The *rgoferd* daemon receives submissions and control actions, the *rqinfod* daemon caches data to answer queries and the *rdispatcher*, the primary focus of this paper, is the decision engine behind core allocation, task-to-node mapping, prioritisation and dispatch.

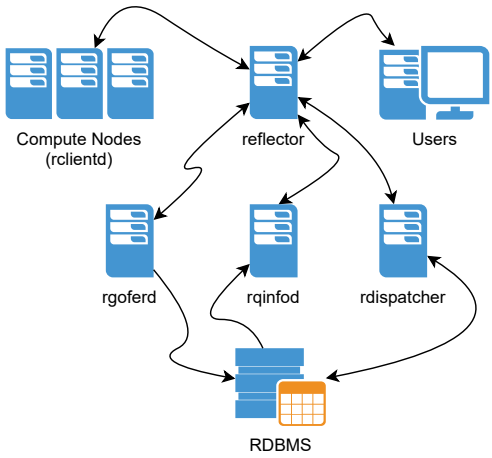

Fig. 1. The Coda Architecture

2.4 Concepts & Configuration

Coda assigns CPUs or cores to so-called pools. Typically each show would have a set of pools assembled in a hierarchical structure (to reflect work type, discipline or priority as they see fit), and a DRM would discuss with the show their requirements and set a core entitlement for these pools accordingly. Any show under-utilising their entitlement of any given pool implicitly permits other pools to borrow from them, leading to speculative execution with a preemption penalty. In other words, tasks are killed and requeued if running speculatively and an entitled pool now requires its cores back to fulfil a demand. Each pool effectively models a priority queue implemented as a heap and tasks are placed according to an ordinal priority. A synthetic structure illustrating a pool configuration is given below in figure 2.

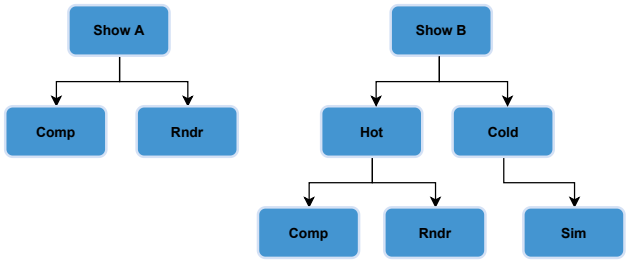

Fig. 2. An example of how CPU Pools can be arranged according to requirement

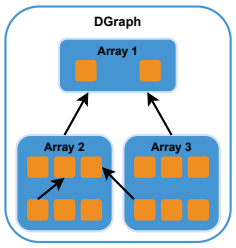

Fig. 3. Illustrating the nested encapsulation of a dgraph (innermost squares represent tasks, arrows represent dependencies)

Coda models jobs as "dgraphs" and these contain *arrays*, which in turn are comprised of *tasks*. An array is often used to represent a frame range, each task thus rendering a frame of the film. Precedent constraints are defined as dependency expressions in *JavaScript* (rather than a DAG directly) between any of these 3 layers. Each instance of an object in a dgraph had a unique identifier, a fully qualified ID being `9876.1.1000`, for example. Metadata attributes, expressed as `key=value` pairs, can also be set at any level, as there exists a hierarchy between them. Thus the lowest level (a task) can inherit values through its ancestry. Reserved metadata keys describe common attributes such as required resources, user, command, dependencies, priority etc. whereas others such as title, shot name or task type are informational only and used in reporting or display data for UIs. Figure 3 to the left demonstrates the job composition.

2.5 Scale

The growth of a render farm at ILM is steady and expands to accommodate increased workload. More shows concurrently, greater complexity (e.g resolution, number of characters in frame), new CG technologies (virtual production, machine learning etc.) all make demands of the farm in ways we've not experienced before. At its peak, the on-premises render farm in London is now 40% larger than 4 years ago when we delivered *Ready Player One*. Cloud bursting has also allowed us to expand capacity flexibly when required - our Singapore studio recently ran at 3x it's normal 30k capacity for a week to meet a deadline. Since introducing Coda across the globe, we even launched an entirely new studio in Sydney which now runs at 50k cores. The sum of ILMs regular on-premise core count now exceeds a quarter of a million and we regularly peak at 40x that for our task-queued cores, which is a lot of work to chew through!

With these sizes, the dimensionality of resources, diversity of workload and pressure of deadlines, we began to experience regular problems in production. As the larger sites scaled their demands, we often found Coda was unable to examine enough tasks during a scheduling run and overall farm utilisation was low.

2.6 Problem Statement

With each facility handling a share of the *ABBA* workload plus their own additional film & TV projects, we'd reached a point where the main scheduling component, rdispatcher, struggled to handle the increased load. This was causing disruption to the service and our users even though we'd not yet reached the

forecasted peak demand. A project was proposed to deal with the fundamental issues we faced, and planned phases to tackle each of them.

Our first challenge was improving our ability to correlate events on a timeline - reports from users or production and the performance of system components, modules etc. We lacked this visibility, and without it, would be unable to easily identify root-causes of performance degradation. This work is covered in section 3. Section 4 then details the cycles[1] we spent identifying the specific bottlenecks and eradicating them. Section 5 presents the necessity of improved tests and how we tackled them, since before now, testing representative production workloads simply didn't exist.

3 Observability

One of our initial challenges was understanding the system performance; where it struggled and why. The very motivation behind this project was to handle increased load (more tasks, more nodes) and we already had at hand, reports of dispatch problems at each site and a modicum of metrics to refer to. However, nothing rich enough to provide greater insight - no key performance indicators (KPI) or engineers' view of the farm and its operational state.

3.1 Logging

To ease the selection of useful log entries that could drive data on a dashboard, we concentrated on rewriting each log line according to a semi-structured format that made extraction and understanding easier. Choosing concise, descriptive text removed previous ambiguity and improved readability and parsing. New messages were also added to expose previously unavailable data. Log lines were extracted and sent, via a handler, in real-time to fluentd [2] for processing before storing in Elasticsearch [1] (ES). The fluentd processing engine was able to generate ES document mappings automatically via the `key=value` pairs present in some of the logs.

3.2 Dashboard

With ES as a data source we now had enough information to build a Grafana [5] dashboard. The aim with this dashboard was to give us, as maintainers of the software and system, useful diagnostic information to quickly narrow down dispatch problems in production. We were able to derive KPIs and, among others, produce resource distributions via binning and a task distribution by priority

[1] An agile development process. See [10]

band for each pool. This latter metric was useful, since one of the primary problems we faced was identifying the flood of tasks to a pool and where in the corresponding heap they were located (guided by their priority). Coda was originally designed to honour absolute (task) order by priority with respect to its pool. A pool's priority queue could effectively be blocked by 100s of 1000s of "undispatchable"[2] tasks, thus starving 1000s of others at a lower priority. This, we found, contributed to the overall low utilisation of the render farm. Figures 4 through 7 provide some small insight into the panels on our dashboard.

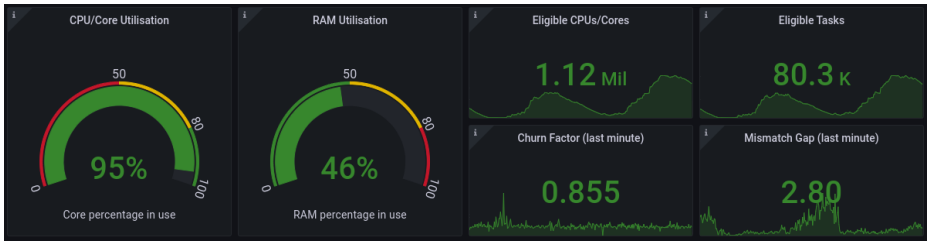

Fig. 4. The high-level overview of an render farm (24h period)

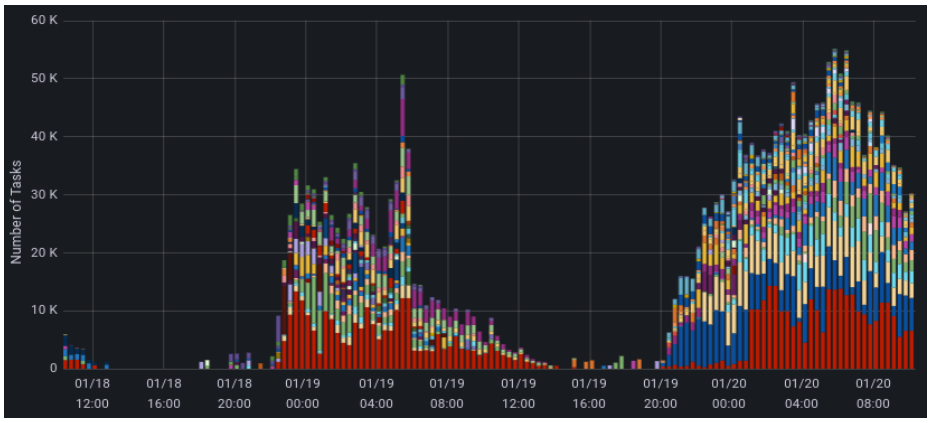

Fig. 5. Distribution of tasks by priority band per pool (pool names omitted)

4 Optimisations

In order to measure the effectiveness of Coda we employ the following metrics:

[2] Tasks unable to be dispatched due to scarce resources, exceeded limitations etc.

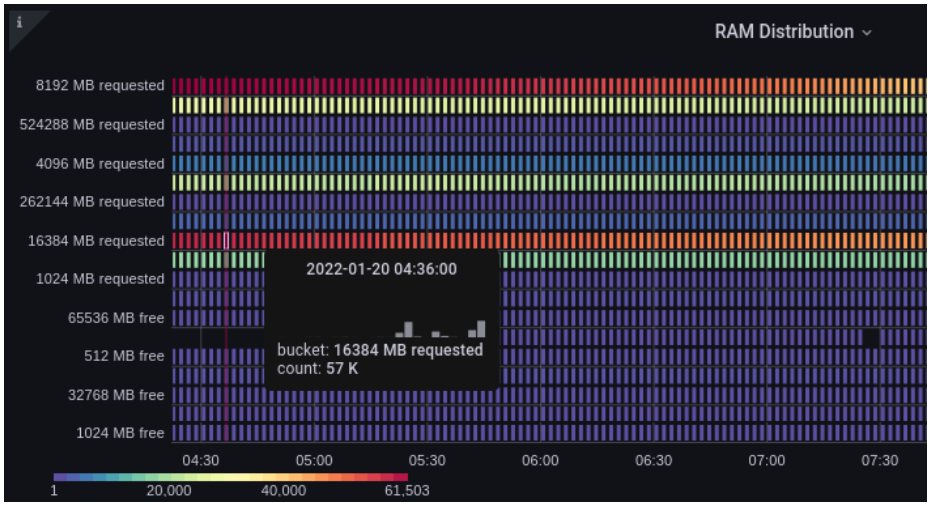

Fig. 6. Requested & Free RAM binned in powers of 2 shown as a Heatmap

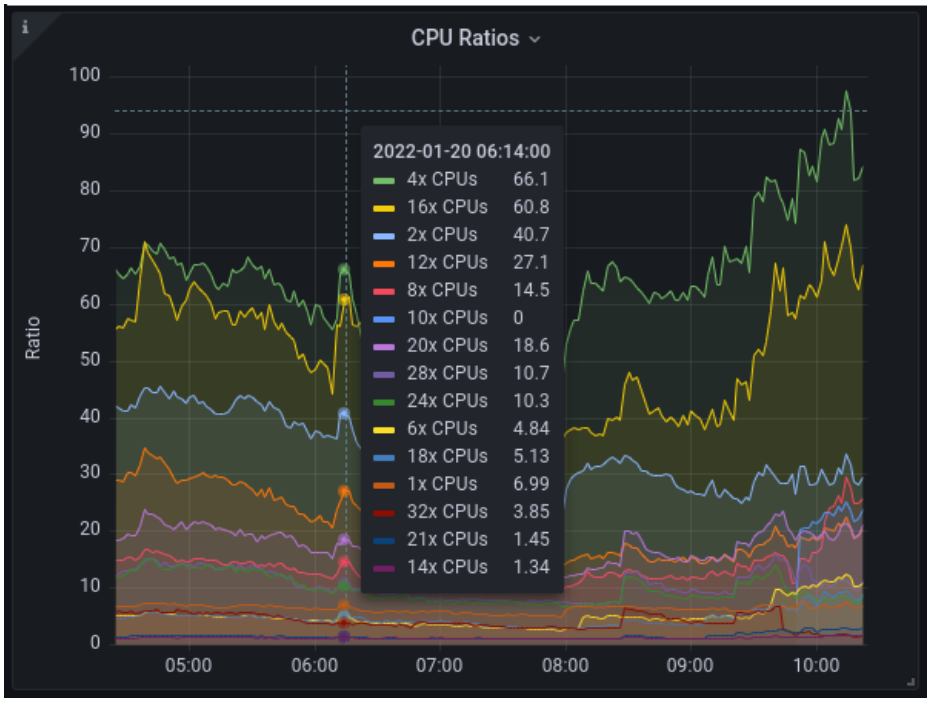

Fig. 7. Distribution of CPUs as a ratio of requested to available. I.e 66x more 4-core slots are required than available to fulfil the queued amount.

- Is the farm busy? Idle machines are a waste of money.
  - This can be measured in real-time.
- Is the farm busy with effective work? Discarding work is a waste of money.
  - This can only be measured accurately in retrospect, tasks may take many hours to complete and they could be killed at any time for different reasons.
- Are pools filling their assigned quotas?
  - This can be measured in real-time.
- Are the hardware resources available on the machine hosting the dispatcher service used effectively? An idle machine is a waste of money.
  - This can be measured in real-time.
- Are high priority tasks completed before lower priority ones?
  - This is complicated to measure but we record submission, start and end times, the number of times the task was examined, the reasons why it was not dispatched, all the status changes, etc. All this information could be used in retrospect to compare dispatch times versus other tasks.

These metrics need to be interpreted in context, there is a big difference in monitored values when there are only one hundred eligible tasks versus when there are hundreds of thousands of them.

We take into account all the metrics when we check the status of the farm and the effect our changes have on its performance. Each of the following subsections report on the major developments required to improve all of these metrics, effectively producing a survey on our recent work. The process as a whole was iterative and relied upon repeated profiling & testing both in production and through simulations (see section 5.4).

4.1 Thread Model

The original implementation had a fixed number of worker or dispatch threads servicing all pools, the main loop being similar to;

```
CPUPool *pool = nullptr;
const bool use_round_robin = true;
while ((pool = select_next_pool(use_round_robin))) {
   Task *task = pool->get_highest_ranking_task();
   if (task != nullptr && examine(task)) {
      Host *host = match_host(task);
      if (host != nullptr) {
         host->manage(task);
         dispatch(task, host);
      }
   }
}
```

This was not optimal for these primary reasons:

1. The process of selecting a pool required access to shared global data that had to be protected by a mutex, which effectively serialised the threads and avoided potential parallelism.
2. Selecting a pool was not free, sometimes it was more expensive to select a pool than to actually examine a task for dispatch, which acted as an unwanted dispatch-rate throttle.

Simply increasing the number of dispatch threads in this model was not providing a noticeable improvement in dispatch rates. We also observed the CPU utilisation of the machine hosting the service was almost never above 40%. These machines had 88 logical cores and they were underutilised. Our hottest metric here was the considerble voluntary context switching performed.

Replace Static Pool Changing the model to have a separate thread per pool rather than a shared pool of worker threads, improved examination and dispatch rates. The few stages of the new thread loop was now:

1. Select the highest priority task from the pool.
2. Examine, match, and dispatch the selected task, if possible.
3. Back to 1

With this model the main serialisation point and the repeated work of selecting a pool were avoided.

Work Stealing The previous change improved our examination and dispatch rates but it did not take advantage of all the CPU cores available on the dispatcher machine when the number of active pools was small. To compensate for this, we implemented a work stealing system for idle pools which worked in the same way the previous thread model worked, by first selecting an active pool to dispatch a task from. With this change we had the best of both threading models, automatically changing from one system to another depending on the number of active pools and some thread-to-pool affinity where possible.

4.2 Parallelisation

Reduce Serialisation Points Despite the thread model improvements, rdispatcher was still unable to take full advantage of all the CPU cores available on the host. Careful analysis and profiling highlighted several global mutexes that were serialising many operations. We removed or avoided several of those global locks and managed to improve our overall parallelism.

Reduce Cost of Critical Sections Several global locks still remained that were required for the effective and safe work of the dispatcher service. In order

to reduce the impact of those locks we reduced the amount of work done in critical sections in order to reduce the amount of time threads were waiting on locks. Some improvements were, for example, to move safe calculations outside of critical sections. We also identified computation that could be carried out in parallel since it did not require access to shared data, or only required read-only access to shared data.

4.3 Redundant Calculations

Reuse *JavaScript* Engines In section 2.4 we highlighted that precedent constraints were implemented as JavaScript (JS) expressions between the graph vertices. We discovered that each JS engine used to evaluate task dependencies were recreated every time those dependencies were checked (which happened frequently). We changed the code to create and reuse one single JavaScript engine per thread, effectively removing the heavy cost of recreating them.

Cache Regular Objects We identified several strings that were reconstructed regularly. By caching these strings we traded off a small increase in memory usage by a massive reduction in string operations, further reducing the load on the memory allocator and avoiding the work required to recalculate those strings.

Introduce 128-bit Integers Tasks are sorted by priority, and the one with the highest value is examined and dispatched first. This ordinal priority was a composite of many individual priorities and the final value computed for comparison was the 40-character string (stored internally as UTF-16) representation of these concatenated priorities. The string operations required to generate and to compare that value appeared high on profiling sessions. Careful examination of the values used to generate that string showed that we could encode the same amount of information as a 128-bit integer, converting only to strings when we had to present that value for debugging purposes. This change allowed the compiler to generate far fewer yet more efficient instructions provided by a big reduction in string operations that otherwise ranked highly during profiling sessions.

Replace Strings with Integers The task state and event system used strings as identifiers which required many string comparisons by the state machine and the event system. The number of different values was fixed and small, which meant that it was relatively easy to convert into integer values. This change again removed a large amount of string comparisons from the profiler sessions.

4.4 Memory Footprint

Many routine bug fixes contributed to an overall reduction in memory footprint and the associated cost or overhead introduced by an allocator continuously trying to manage memory. There were two modifications we made however, where the significance is worth highlighting. Eventually, we reduced the mean memory consumption by 50%.

Share a Fixed State Hierarchy A finite state machine was created for each task, and with hundreds of thousands of tasks active at each single run, and millions more inactive but in cache, meant a significant amount of memory allocations were duplicated for this graph. Upon investigation we concluded that the state graph was fixed and constant during the runtime of the service and that we could generate the graph at startup time and to share it with all the tasks. The original implementation of the state machine was very generic, allowing for dynamic and variable state graphs, but the actual usage of the state machine was actually static once the service had started. This change saved millions of memory allocations, significantly helping to reduce the load on the memory allocator.

Reduced Temporary Objects Several components of the dispatcher service were creating too many temporary objects, often unnecessarily. Carefully rewriting the code to reuse or reduce the number of temporary objects improved the load on the allocator.

4.5 Memory Allocator

Our profiling sessions often showed the memory allocator (tcmalloc [4]) as responsible for up to 30% of the CPU usage. After some investigation and testing we identified a more suitable allocator (mimalloc [7]) which now rarely exceeds 1% of the CPU usage according to the profiler. The previous allocator was not suitable for the amount of CPU cores and threads and the memory allocation patterns our service required.

4.6 Stability

Serialise State Changes The original implementation allowed task state modifications from multiple threads at the same time. There were mutexes attempting safe operations but there were several race conditions that contributed to abnormal operations. For example, dispatching the same task twice to two different clients at the same time or task metrics being accounted for on the wrong pool. By serialising task state modifications we removed the race conditions and improved the behaviour of the system.

Generic Bug Fixes There were logical bugs that affected the correct behaviour of the service. For example the recalculation of available CPU cores was returning negative values for overallocated clients, and those negative values were wrongly affecting the cache structures used to identify clients that provided the required amount of resources for a task.

4.7 Asynchronous Events

Simplified and Streamlined Event System The Qt [9] event system was used heavily in the original implementation but it's generic applicability was not performant enough for our needs. We recorded latencies in excess of 1 minute between when an event was emitted until it was processed, due to the sheer amount of events we were generating (sustaining peaks of 400,000 events per minute), and due to the cost of processing each individual event. By implementing a much simpler and specific event system we reduced both CPU and memory usage and greatly reduced event processing latency.

Simplified and Streamlined Timer System As part of the implementation of the new event system we also implemented our own timer system highly integrated with the event system and tailored to our simple and specific requirements, further reducing memory and CPU overhead.

4.8 Networking

Reduced Data Copies We identified a redundant data copy every time we received a message from the network. We receive hundreds of thousands of JSON-encoded messages per minute, each averaging 2.3KB. Removing this redundant buffer, among a few other trivial optimisations, increased our throughput by up to 50% at times.

Remove Shared Sockets We were using a single socket for all the messages sent and received, and because of the way messages were encoded we needed to serialise access to that socket, otherwise headers and bodies of different messages could interleave and corrupt the communication between the endpoints. Because the number of threads using the socket was small it was feasible to create a separate socket for each thread which allowed us to parallelise network operations more efficiently.

4.9 Work Distribution

Spread Dependency Checking Evenly The system often has hundreds of thousands of tasks waiting on others to complete. Depending on the job type,

these dependency checks can be quite complex (E.g. FX simulations). The dispatcher service employs several worker threads constantly running those dependency checks and originally it was observed that some threads had 10 times the work than others, which resulted in some dependencies taking a lot longer to be satisfied than others just because the thread handling the checks was extremely busy. We identified that the key used to distribute the work among the worker threads was not evenly distributed because it only depended on the ID of the dgraph owning the task (again, see Fig. 3). After comparing several alternative hash algorithms, we settled on choosing the popular FNV-1a [3] and switched to combining the fully qualified ID. This resulted in a near perfect balance of work across available threads.

Revised Thread Count Each subsystem of the dispatcher service has its own number of dedicated threads, and in some cases the number of worker threads too small, this was identified by the latency of processing work items. Simply increasing the number of worker threads on those subsystems improved the responsiveness of the whole system. We now routinely examine the CPU usage of each worker thread in order to identify subsystems that may require additional resources.

4.10 Waste Reduction

Ordered Preemptive Task Selection When a pool has reached its allocated farm quota it is still allowed to dispatch work to clients but that work is deemed speculative, and hence can be preempted at any point by a regular task entitled to run. By ordering speculative tasks according to their running time and by selecting the one(s) with the least amount of work done, we reduced the amount of wasted core-hours on the farm.

Ordered In-place Promotion (IPP) When the dispatcher has the option to promote a speculatively running task over dispatching a new task of the same shape and belonging to the same pool, it does so to reduce "lost" core hours (a form of waste). To further improve the selection of these otherwise preemptable tasks, we now order by runtime in decreasing order ensuring to promote the longest-running task first.

Recovery of "ghost" Tasks Sometimes clients become unresponsive and after a specific amount of time. Without hearing any status updates from these hosts, we consider the tasks running on them as "dead", and we proceed to dispatch the task to another client. But in many cases the lack of updates was due to some temporary reason such as network congestion, machine overload etc. and the client is still functioning correctly and the tasks are still running. We are now

able to identify these situations and to recover those dead tasks that are still running, hence avoiding the wasted effort of dispatching that task to another machine and starting over.

5 Simulations & Testing

An obstacle often inhibiting the adoption of new or upgraded critical software in production is sufficient and relevant testing. Can we with good confidence guarantee that *a)* we won't degrade the service and *b)* achieve our objective(s) such as handle increased load?

5.1 Coda In A Box (CIAB)

The system components listed in figure 1 run across a variety of powerful host systems in production. Developers don't have this luxury and bringing up a system and all its components including several compute clients was cumbersome. To simplify this step we adopted a container approach early on, running all components sandboxed managed via utility scripts. This container could then also be run on a single powerful machine sufficient to handle greater loads, although running 1000s of rclientd processes remained impractical.

5.2 Integration Tests

To identify regressions we wrote short integration tests that create tasks using the public API and verify that they run successfully within a time limit. We test a fair amount of the internal functionality through these tests (though not everything) - enough to detect obviously broken builds. We run the integration tests in a CI/CD fashion, as a vanguard before any other test.

5.3 Saturation Tests

For more comprehensive testing we created saturation tests. We create 280k tasks with dependencies between them and distributed among 50 different pools. Every minute we issue thousands of random changes to those tasks, such as priority increments or pool migrations. These task modifications helped us to identify and subsequently fix many race conditions. The tests may take up to one hour to run and require a machine with dozens of cores in order to run properly. We run these tests extensively during development when profiling and are good indicators of improved or degraded performance.

5.4 Simulation

At the heart of the Coda system is the reflector (see Fig. 1) and a rich set of messages pass through it - every message necessary, in fact, to replicate a production workload targeted at a separate test system. We developed "refractor", a tool that simulates a render farm given the recorded input stream of another.

In capture mode it simply listens, as a read-only client, to a source reflector (from a production system) for all the messages necessary to rebuild jobs, mimic compute nodes and gather configuration data. This data, once processed, is then serialised to disk ready to replay on a separate test system at will (though there is also a real-time mode).

Refractor can replay recorded input streams, on a test system. It submits the jobs exactly as a user would retaining structure, resource requirements, priority etc. It also creates and configures as many compute nodes as recorded ready to accept the tasks scheduled by the test dispatcher. However, each node is an object in code rather than a real machine. Tasks don't execute the command a user intended but rather sleeps for the known duration, which was encoded into the stream as completed tasks were observed.

This new tool gave us the ability to simulate a render farm at the scale we expect in production. Moreover, it also gave us representative jobs and compute node configurations such that we were able to better simulate the behaviour of a production system under the same conditions (e.g. artificial user limits, resource sharing & scarcity, error rates etc.).

6 Results

We provide results measured from simulations through refractor plus metrics recorded in production. We also demonstrate the improved resource utilisation of the host machine the dispatcher runs on.

6.1 Simulation & Saturation

Our trace data replayed through versions old and new (denoted via their semantic versioning) consistently demonstrate improved performance when compared. Figure 8 shows an overall better farm utilisation. Repeated tests with different daily traces from 3 sites (London, Sydney and San Francisco) all show the same overall increased throughput. We found that the refractor tool is also limited by the number of "in-flight" ascynchronous events *Python 3* can handle. Some of our larger simulations are constrained by this and as such, may reconsider its implementation in the future.

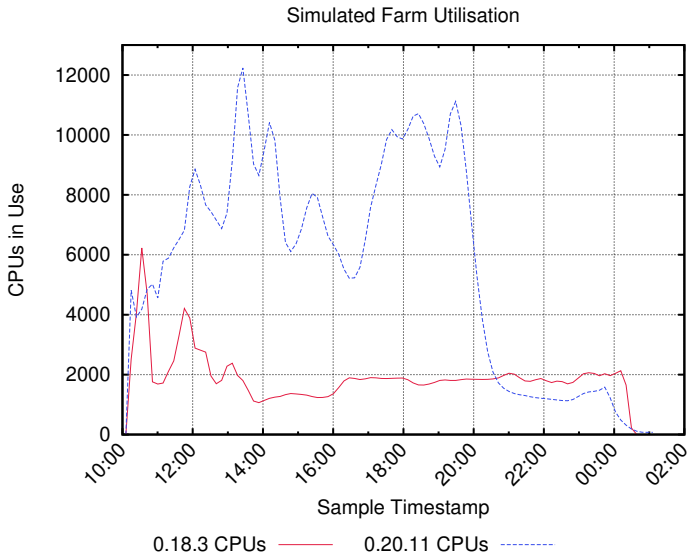

Fig. 8. CPU Utilisation of the refractor render farm using a 15h trace from our San Francisco studio

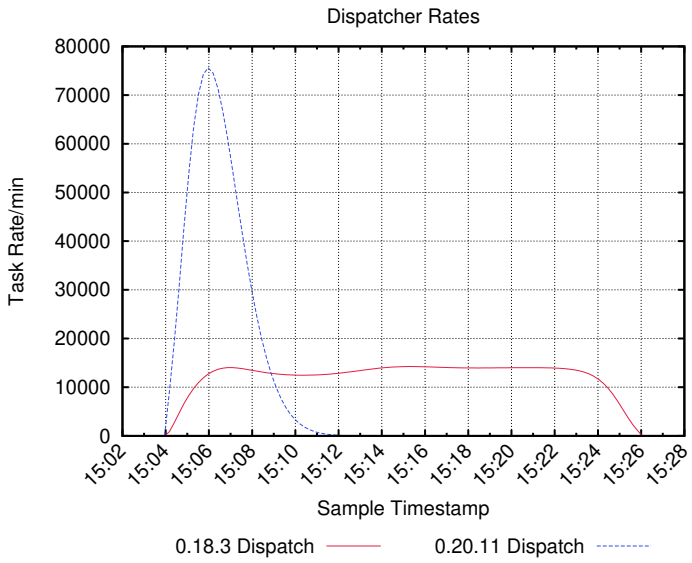

Fig. 9. Task Dispatch Rate for a Saturation Test with 1k Compute Nodes

Figure 9 shows improved task handling by the dispatcher as well as a reduced makespan. It clearly illustrates the efficiency gains of the newer system, and these results remain consistent even under increased load (i.e. greater rclientd counts).

6.2 In Production

Using our dashboard (see 3.2), we were able easily extract a pair of 3 week averages of some simple metrics that demonstrate the effectiveness of our work in production. The averages were taken while the render farm was handling similar workloads from the same set of shows a month apart. Table 1 displays a subset of the quantifiable data we measure to compare the old vs. new dispatcher. Examination is when Coda first pulls a task off a pool's heap, preparing to attempt resource matches etc. Dispatch indicates a successful assignment of a task to a host. Turnover is a measure of tasks completing (regardless of success or failure).

The examination rate illustrates a typical 20x improvement in throughput - the new threading model, reduced stalling (less lock contention) and more efficient task comparator allows the dispatcher to examine more tasks, quicker.

Table 1. Average rates (per minute) for London from November-December 2021

Metric	**0.18.3**	**0.20.11**
Task Examination Rate	2.01k	43.8k
Task Dispatch Rate	953	2k
Task Turnover Rate	1.12k	2.51k

The dispatch rate has more than doubled owing to the new allocator, rewritten events system & balanced work handling. Similarly the turnover rate, which will closely follow the dispatch rate, has improved due to these same optimisations plus the more economical message handling.

Note that no two days are the same on a production render farm, so naturally numbers and rates etc. fluctuate. It is therefore difficult to compare the two versions reliably in production without the guarantee of the underlying workloads remaining identical (i.e. in a trace-based simulation). However, every care has been given to select periods as close as possible and to calculate the mean over a large enough window.

6.3 Server Resources

Here we report briefly on the more efficient use of the server resources following the software changes we made to the dispatcher. Section 4.1 introduced the changes we made to the threading model. This change, coupled with attempts to reduce lock contention to a minimum, resulted in a 50% reduction in context switching and a corresponding boost in on-CPU time. We're finally exceeding

the 40% CPU utilisation threshold mentioned in section 4.1. This efficiency gain is reflected in the examination rate given above in section 6.2. Figures 10 & 11 respectively, illustrate clearly the improved CPU utilisation for two separate 7 day periods for a similarly loaded system.

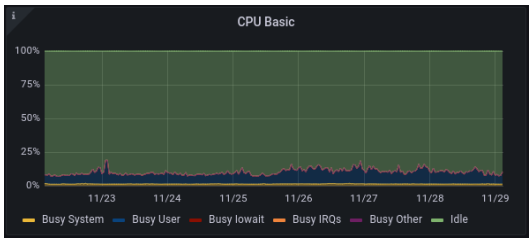

Fig. 10. CPU Utilisation of the server under rdispatcher 0.18.3

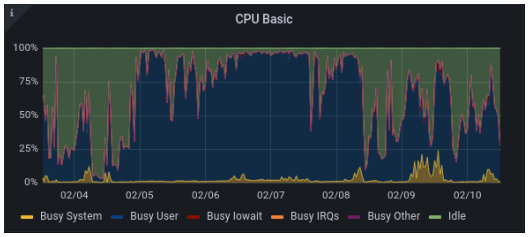

Fig. 11. CPU Utilisation of the server under rdispatcher 0.20.11

7 Future Work

This paper has concentrated only on the changes necessary to improve our current production scheduler and makes no mention of any research into alternatives. However, we have experimented with several ideas we wish to revive, to further maximise our general render farm utilisation.

We're keen to quickly compare common benchmarks such as the minimisation of makespan, mean user wait-time, resource fragmentation etc., when trace data is run through different scheduling policies or established dispatching rules from [8] (e.g FCFS/ERD, SJF, LRPT, WSPT etc.). Similarly, we wish to evaluate the different packing strategies available to us such as NF, BF, FFD etc. as well as our own proprietary algorithms. To this end we'd begun work on a pluggable/-modular framework where combinations of these can be composed quickly and tested offline without the need of production systems.

Many scheduling policies require some knowledge of expected task runtimes and we've already disclosed that rarely do we have this information available as decision criteria. However, in order to evaluate the suitability of deadline scheduling with preemptive backfill through the aforementioned framework, we must first tackle this problem. To this end, we had some success experimenting with simple attribute-based grouping and unsupervised clustering of tasks and, through ARIMA and EWMA, forecasting runtimes for the tasks categorised by these clusters. The framework again supports pre-processing refinement stages such as this prior to the main scheduling run.

We hope to resurrect these efforts in R&D in the near future and bring them closer to conclusion and, ideally, production.

8 Acknowledgements

The authors would like to thank Andre Prado (ILM) and Tommy Burnette (ILM) for their support and feedback, and Kevin Constantine (WDAS) and Graham Whitted (WDAS) for their valuable insight and reviews of the paper, not to mention their considerable contributions to Coda over the years at WDAS - we are grateful for your collaboration! We'd like to also thank Jason Cox (The Walt Disney Company) for his encouragement and again, review. Finally, we'd like to also thank Christian Dahlberg (ILM) for his contributions to the larger project.

References

1. Elastic: Elasticsearch, `https://www.elastic.co/elasticsearch`, an index+search analytics engine
2. Fluentd: Fluentd, `https://www.fluentd.org`, an open source data collector
3. Fowler, G., Noll, L.C., Vo, K.P.: Fnv-1a, `https://www.ietf.org/archive/id/draft-eastlake-fnv-17.txt`, a non-cryptographic hash function
4. Google: Tcmalloc, `https://github.com/google/tcmalloc`, a memory allocator
5. Grafana Labs: Grafana, `https://grafana.com/oss/grafana`, an open source web-based visualisation platform
6. Industrial Light & Magic: ObaQ, `https://www.linuxjournal.com/article/6783`, a public disclosure of the ObaQ/DOALL Render Farm Queuing System
7. Microsoft: mimalloc, `https://github.com/microsoft/mimalloc`, a memory allocator
8. Pinedo, M.: Scheduling Theory, Algorithms and Systems, chap. 7. Prentice Hall, 1 edn. (1995)
9. Qt Group: Qt, `https://www.qt.io`, a cross-platform application development framework
10. Singer, R.: Shape Up - Stop Running in Circles and Ship Work that Matters. Basecamp (2019), `https://basecamp.com/shapeup/webbook`
11. Walt Disney Animation Studios: Coda, `https://www.disneyanimation.com/technology/coda`, a public disclosure of the Coda Render Farm Queuing System