Creating a system based on a Data-Oriented paradigm that compliments an Object-Oriented Workflow.

Alexander Stephen Allman
Alexander2.Allman@live.uwe.ac.uk
Supervisor: James Huxtable

Department of Computer Science and Creative Technology
University of the West of England
Coldharbour Lane
Bristol BS16 1QY

Abstract

This project is designed to allow developers to keep an Object-Oriented workflow whilst still gaining some of the performance benefits from the Data-Oriented paradigm. This is done through leveraging a custom memory allocator for storing related data spatially local and job system to thread permutations on the data.

Keywords: Memory allocator, Job system, Threading, Data-Oriented

Brief biography

After hearing about Unity's new DOTS system (Meijer, 2019), I decided to read into Data-Oriented design as I wasn't previously aware of the paradigm. Seeing that the paradigm stores and refers to data in a vastly different manner it took a bit of rearranging to how I approached coding problems. From this I wanted to make a system where people who utilize an Object-Oriented workflow can still gain some of the benefits from a Data-Oriented one whilst easing them into some of the harder concepts of Data-Oriented design. This project has helped me expanding my knowledge of Multi-Threading and more advanced memory allocation in C++ and a postmortem can be found on my portfolio at: https://xalexxd.github.io/

How to access the project

To access the project please go to the projects GitHub page at: https://github.com/xAlexXD/CTP-DOD-in-OOD

1. Introduction

The aim of this project was to create an interface in which a developer who is versed in Object-Oriented design could still gain some of the performance benefits from a Data-Oriented workflow. This has been achieved through a custom Linked List Free List allocator (Trebino, 2019), a memory manager to interface with the allocator and a Job system to handle threading of tasks to be processed per update. This project has been designed to be plug and play so that all a developer needs to do is drop the source files into their project, utilise the memory manager for variable initialisation and the job system to process permutations on the data. After this setup, performance benefits from the data locality and parallelised processing of the data become apparent. Developing with this system will help developers unfamiliar with the Data-Oriented paradigm understand the reasons why it is more performant and ease them into understanding the paradigm; acting as a bridge for the developer to deepen their knowledge further.

This project delivers the following:

- A Free Linked list allocator to store data
- A Memory Manager to interface with the allocator and ensure that related data is stored spatially local to each other.
- A Thread pooled Job system which acts as a simple to use interface to process tasks in a multithreaded manner.

2. Practice

2.1 Project Outcomes

The project is built up of three main components. Firstly, the free list memory allocator which handles requesting a large block of memory from the OS and storing data sent to it by splitting the large block into smaller chunks. Secondly, the memory manager which closely interfaces with the memory allocator and finally, the job system which handles threading permutations on data.

The memory allocator was the first system to be developed and tested. Once in place, both the memory manager and job system were built and improved in tandem as the project evolved.

2.1.1 Memory Allocator

The Memory Allocator is based from a free linked list design. This was chosen because searching through a list of already available blocks is much quicker than traversing the entire linked list for a free block. When just using a normal linked list, every time a new allocation is to be made the system will have to potentially traverse the whole list which is O(n) where n is the size of the linked list (Taylor, 2019). By utilising a free linked list, the search stays as O(n) but the search space is significantly smaller. This is due to the separate list only containing blocks available for allocation. In Figure 1 a visual of this can be seen where the green blocks that are free for allocation are connected, skipping checks on already allocated red blocks.

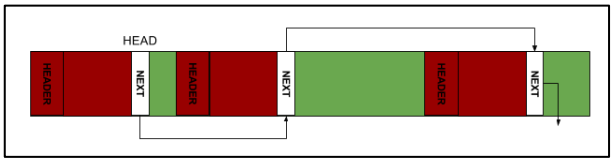


Figure 1 - Visual of a free linked list allocator, where free allocations are directly connected saving traversal of the entire list (Trebino, 2019).

Efficiency and flexibility of allocations were also improved via the splitting and coalescing of blocks, seen in Figures 2 and 3 respectively. When allocating a block of memory, the allocator automatically splits the block at the size requested and makes a new block from the remaining size. This means that no space is wasted for each allocation as only the requested amount of memory is being used. In the same vein, when a block is freed from being used the allocator checks whether the next block is free and If so, then merges the blocks into one larger block. This helps reduce the size of the free list for quicker search and allows a chunk of data to

be stored that was larger than the two single blocks but smaller than their sum.

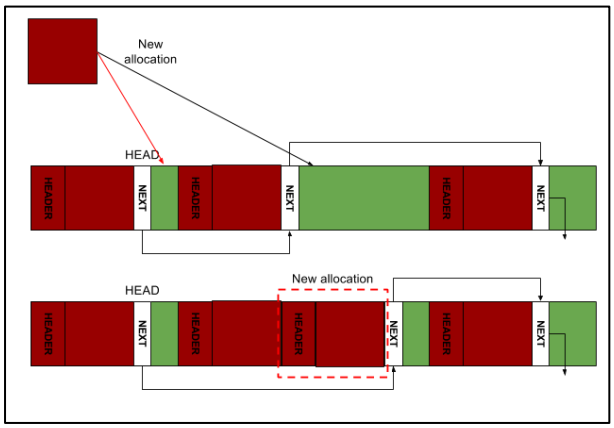


Figure 2 - Splitting of a larger block into the required size for the allocation and making a new block from the smaller one (Trebino, 2019).

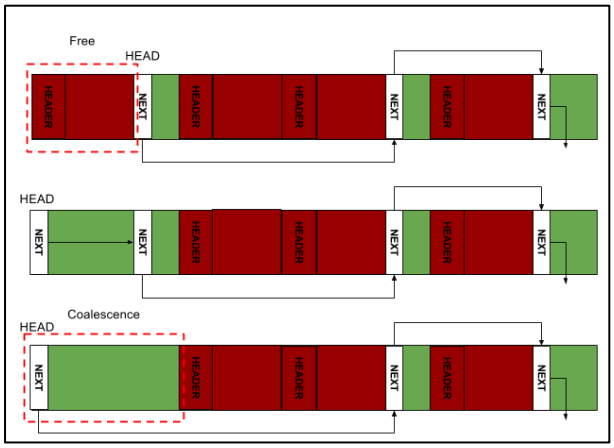


Figure 3 - Coalescing of two free adjacent blocks (Trebino, 2019).

2.1.2 AllmanVariable

The AllmanVariable is a custom templated class which acts as the interface for the developer to communicate with the memory manager. It takes a pointer to a memory manager and a hash id on construction and with this receives a pointer from the memory manager to store its data in. The hash id is used to determine what data is being stored in terms of the problem domain.

This allows the memory manager to know which variables are related and store those variables together in contiguous memory. This class is also used to interface with some of the memory managers stored meta-data. This meta-data includes how many allocations there are in a block, the max it can hold, the start pointer to the block and the stride to jump between elements. The structure of this can be seen in Figure 4.

```
struct ExistingBlock
{
    size_t    hashId;
    void*     dataPointer;
    int       currentCount;
    int       maxCount;
    size_t    stride;
}
```

Figure 4 - The structure of an existing blocks meta-data.

For example, if you wanted to go over all variables related to one hash id, the developer can quickly get the Base pointer of the memory block, how many items are currently stored in it and the stride to loop through all elements stored in the contiguous block of memory. Initialisation and iterating over a set of variables can been seen in Figures 5 and 6 respectively.

```
Health = AllmanVariable<float>(
            _memManagerPtr,
            AllmanHash("HEALTH"),
            _defaultHealthVal
);
```

Figure 5 - Initialisation of an AllmanVariable.

```
float* base = Health.GetBasePtr();
for (int i = 0; i < Health.GetCount(); ++i)
{
    float* dataPtr = base + i;
    *dataPtr = 2.5f;
}
```

Figure 6 - Looping through a set of locally stored variables via the base pointer.

2.1.3 Memory Manager

The memory manager is the owner of the memory allocator and manages storing data in blocks of memory received from the allocator. When initialising a variable through AllmanVariable with a hash, the memory manager does a few things.

Firstly, it checks to see if there is already data stored related to that hash id. If there is then it appends the new variable onto the existing set of data within that allocated block and increments the counter to reflect the new stored data. Finally, it returns a pointer for use to refer to that variable, which the AllmanVariable stores in a private member variable for quick access. This process is described in pseudocode in Figure 7.

```
if (block already exists with hash ID)
{
    • Get the new pointer by using the base ptr + current count.
    • Add 1 to the current count.
    • Return the new pointer.
}
```

Figure 7 - Check for existing block when initialising an AllmanVariable.

If a memory block has not been allocated for the inputted hash id, then the manager requests a new block from the memory allocator. The allocator returns the base pointer to this block and then the manager sets up some meta data for the block which can be referenced by the variables. Pseudocode for this can be seen in Figure 8.

```
if (block doesn't exist with hash ID)
{
    • Request a new block from the memory allocator.
    • Create ExistingBlock meta-data using the returned pointer as base, setting the hash ID from the request and setting the Count to 1.
    • Return the base pointer as it's the first allocation.
}
```

Figure 8 - When there isn't an existing block matching the inputted hash ID.

All members and functions within the memory manager are private, with exception of the constructor. To interface with the memory manager the developer must go through AllmanVariable. To allow AllmanVariable to interface freely with the memory manager, it is a friend class of it. Whilst this does break OOP rules of encapsulation, the AllmanVariable is so intrinsically linked on the memory manager to function that this is a reasonable use case for the keyword. This way AllmanVariable can interface with the memory manager where necessary, whilst the developer can only access the initialisation of the memory manager itself and the public facing functions of the AllmanVariable. This was a key concern from the beginning of development as it was desired for any developer to be able to pick up and use the systems. Allowing the developer too much access to the back-end features would not only be a poor design choice in terms of the systems robustness, but also overwhelming to learn.

2.1.4 Job System

The job system allows the developer to easily thread tasks. This is done through giving the job system pointers to functions they would like ran on a thread. These function pointers are then processed on pooled threads owned by the job system. A visualisation of this can be seen in Figure 9.

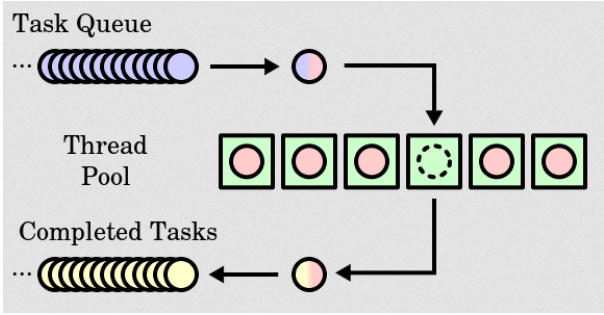


Figure 9 - Visualisation of the Job system. It receives tasks from the developer, processes them via the thread pool then once complete, frees the thread for more tasks (Trebino, 2019).

By default, these jobs are processed unordered but if the developer desires for certain jobs to be completed before others they can put a priority on the job when assigning it to the job system. It's recommended, however, to have as many unordered jobs as possible to reduce hang up on waiting for other jobs to be completed.

The Job system works on a two-step process. Firstly, the job system receives any jobs to be processed by the developer with their accompanying priorities in the form of a Job (as seen in Figure 10). Once all jobs have been received and added to the job systems queue, all jobs are then processed. Ordered jobs get processing priority to remove jobs with dependencies as quickly as possible, whereas, unordered jobs use threads as and when they are available.

```
Class Job
{
    std::function<void()> functionPtr;
    JobPriority priority = UNORDERED || ORDERED;
    int priorityOrder;
}
```

Figure 10 - Structure of a Job.

Whilst jobs are limited to only taking void() function pointers, std::bind can be leveraged to allow the processing of functions with parameters as seen in Figure 11. One drawback is the inability to have results return from a function and potentially feed into a following job. The developer would instead need to leverage atomic variables stored in a mutual class and the job priority system.

```
void FunctionWithParameters(int 1, int 2);
std::function<void()> functionPtr;

functionPtr = std::bind(
        &FunctionWithParameters,
        integer1,
        integer2
);
```

Figure 11 - Binding the function pointer with parameters for use with the job system.

If the developer knows they are repeatedly going to call the same job every update, they can create a cached version of the job for the job system to run every update. This would save on the overhead involved with creating a new job, function pointer and binding a function to it every update.

The job system also adapts how many threads it will pool depending on the hardware it is running on. It uses std::thread::hardware_concurrency() to get a start point based upon how many logical threads the machine possesses. From here using the thread count, threads are initialised with a test function made to process enough data to force context switching between the threads. The time taken to process the permutations on the data is taken and is compared to the previous runs test. If there is no significant drop in performance, the test is run again but this time with more simultaneous threads. This is repeated until there is a substantial increase in time taken to process the data permutations. At this point the upper limit of simultaneous threads has been found. Once found, a ratio of that thread count is used to initialise the thread pool with. This ratio is determined by how taxing the developer wants the software to be. This can be seen in Figure 12.

$$ThreadCount = floor(MaxCount * IntensityMod)$$
Where $IntensityMod = 0.4, 0.6$ and 0.9
for $low, medium\ and\ high$ CPU intensity respectively.

Figure 12 – Equation used to determine thread count for the Job System.

2.2 Project Reflection

The research phase undertaken for the project was vital. Since previous knowledge on the topics were sparse other than a genuine interest in how the Data Oriented paradigm functioned; research and testing various implementations were required to get a solid basis to build from.

Supervisor advice was also key to the project's success. Having a domain expert to discuss ideas and potential routes to follow for implementations sped up early iterations of the project significantly as it assisted in avoiding going down paths that would lead to a dead end.

For the memory allocator researching helped narrow down on different implementations of allocators and which would be the best fit for the project. A balance had to be struck between raw performance and flexibility. Whilst a linear allocator would have the best performance (Trebino, 2019), it also is the worst in terms of flexibility when freeing allocations. Future improvements to the implementation were also made apparent such as switching from a Free Linked-List to a Segregated Free Linked-List which would have reduced the search space when looking for free blocks further.

The Job system benefited from consistent research into new concepts throughout. From the beginning where semaphores and their implementations were studied to be able to dictate an order of processing jobs; to learning how std::future() and std::promise() works to ensure detached threads that are part of the thread pool can safely be terminated on exit.

2.3 Project Evaluation

Throughout development there were many challenges overcome. Some of which were overcome alone through time away from coding to plan out an effective solution; and others required new knowledge.

The initial implementation of the Job system didn't leverage thread pooling; it just created new threads to run jobs on every tick and waited for those threads to join before continuing execution. This set and forget solution did not work with thread pooling as threads needed to be consistently polled to see if the job that was being processed was completed in order to assign more jobs or continue program execution. This resulted in a large refactor of the system which involved creating a pool of detached threads which were contained in their own class. This interface was then used by the job system to allow it to get the state of the thread and give it new functions to process if it's free.

The first version of the memory allocator also had a fatal issue. It began as a simple linear allocator to help reinforce the fundamentals of allocators and quickly debug issues on a simpler system. When allocating new blocks of memory there was a problem where blocks were being created halfway through the previous block, resulting in the whole system breaking down and leading to undefined behaviour. This was quickly fixed after a discussion with the supervisor and learning that the pointer arithmetic being performed in the memory allocator was incorrect. Once this was fixed, offsets were calculated correctly, solving the problem. This knowledge was then carried onto the creation of the Free Linked-List allocator and part of the Memory manager implementation.

3. Discussion of outcomes

To investigate the performance benefits of the implemented systems, four tests were created. A baseline test which used default allocations and ran on a single thread (No_systems). A test which utilised the memory manager and allocator on a single thread (Just_Mem). A test which utilised the job system (Just_Job) with default allocations and finally, a test which used both the memory manager and job system in tandem (All_Systems). Each test ran on an increasing number of NPCs which had their variables adjusted every tick to simulate normal gameplay. The time each tick took to process was taken and an average from twenty of the results was used to create the graphs in Appendix C and D.

These tests were run on two different systems. A desktop running a Ryzen 2700x clocked at 3.95Ghz with 16gb of RAM clocked at 3200Mhz and a laptop running an i7-8565U clocked at 1.8ghz with 16gb of RAM clocked at 2400Mhz.

The graphs show that when the memory manager (Just_Mem) and job system (Just_Job) are used alone, they have a significant performance boost over the No_Systems counterpart. When both systems are used tandem in the All_systems test, the performance delta in comparison to No_systems only grows further. This aligns with Longbottom, Papadogiannakis et al and Cruz et al (2017; 2012; 2014), where both multi-threading and keeping memory spatially local results in performance gains. By leveraging both of these ideas together, All_systems had greater benefits than leveraging a single idea at a time (Just_Job or Just_Mem).

Whilst the performance benefits are clear from the data shown in the graphs, there are some drawbacks to these systems that the developer will have to work around to gain those benefits.

Firstly, even though blocks of allocations are organised in a free linked list; the way the memory manager stores variables in these blocks is linear in nature. This means that once the developer has allocated a variable with a given hash; the only way to get that space back for that allocation would be to clear the entire block related to that hash. This negative can be circumvented however by utilising an object pooling system (Kircher et al, 2002). Not only does this save the cost of instantiating memory on run time like a typical object pool, but also guarantees that the related variables are spatially local through using the memory manager.

Secondly, even though variable initialisation has been streamlined significantly since the early iterations of the project, it still is not as simple as using the new keyword (CppReference, 2020). This results in a slight learning curve for the developer and a bit of work converting existing code to use the memory manager.

Finally, the developer using the systems also must make a change in mindset to leverage the full performance gains from these systems. Whilst data is can still interacted in an Array of structures manner, the performance benefit is gained through processing permutations on the data in a structure of arrays manner (Amanda, 2019; Intel, 2018; Fabian, 2018). For example, instead of querying each object for their health and giving them regen on that health per tick; the developer should scrub forwards from the base pointer of the health memory block and apply the same permutation to all health values. This would save polluting the cache lines with unnecessary data from the class and is one of the key benefits of Data-Oriented Design (Llopis, 2009).

By adopting this method of problem solving, it also allows the developer to leverage the job system much easier. Once the developer has learnt how to split their problems into generic tasks for a set of data it is only one extra step to make this task a job and hand it to the job manager to thread. If not all problems can be solved in this manner however, a union of both methodologies can be used; processing all generic tasks using the job system then specific tasks afterwards. Although it is recommended where possible to avoid this as almost all performance benefits of the memory manager derive from performing mutations on the data in a SoA manner.

The design of this project differs to the status quo as it is believed by Data-Oriented programmers that Object-Oriented programming is holding software back by not focusing on the hardware the code is running on and making the problem domain the focus rather than the data itself (Fabian, 2018; Llopis 2009). This arguing over which paradigm is superior causes unnecessary division and hinders progression in the development of software architecture. The aim of this project is to not show one paradigm to be superior to another, but, to act as a bridge for developers to explore a different paradigm and utilise it where appropriate. Data-Oriented design requires a significant shift in how problems are considered and solved (Llopis, 2009), but the performance benefits shown in this project make exploring this paradigm a more than worthy endeavour.

4. Conclusion and recommendations

In conclusion the Data-Oriented inspired systems developed in this project resulted in a significant performance benefit over to a typical non threaded, Object-Oriented counterpart. Whilst

forcing Data-Oriented design into an Object-Oriented system isn't practical as the paradigms are vastly different. Software that utilises both purely Object-Oriented and purely Data-Oriented systems in tandem could be. This project showed the performance benefits that can be gained through Data-Oriented design in a way is familiar to Object-Oriented developers. The aim was then to inspire these developers to further research Data-Oriented design to make the paradigm more known and have more hands working towards improving software architecture in the future.

5. References

Amanda, S. (2019) *Memory Layout Transformations.* Available from: https://software.intel.com/en-us/articles/memory-layout-transformations [Accessed 16 April 2020].

Cruz, E.H.M., Diener, M., Alves, M.A.Z., Pilla, L.L. and Navaux, P.O.A. (2014) Optimizing Memory Locality Using a Locality-aware Page Table. *IEEE 26th International Symposium on Computer Architecture and High Performance Computing* [online]. 26 (1), pp. 198-205. [Accessed 16 April 2020].

Cuervo, T. (2019) Big O Cheat Sheets. *Algorithms & Data Structures*[blog]. 21 March. Available from: https://cooervo.github.io/Algorithms-DataStructures-BigONotation/index.html [Accessed 16 April 2020].

C++ Reference (2020) *new Keyword.* Available from: https://en.cppreference.com/w/cpp/language/new [Accessed 16 April 2020].

Fabian, R. (2018) *Data-oriented Design: Software Engineering For Limited Resources and Short Schedules*. United Kingdom: Richard Fabian.

Intel Software (2018) *How to Manipulate Data Structure to Optimize Memory Use on 32-Bit Intel Architecture.* Available from: https://software.intel.com/en-us/articles/how-to-manipulate-data-structure-to-optimize-memory-use-on-32-bit-intel-architecture [Accessed 16 April 2020].

Kircher, M., Parshant, J. (2002) Pooling. Siemens. Available from: http://www.kircher-schwanninger.de/michael/publications/Pooling.pdf [Accessed 16 April 2020].

Llopis, N. (2009) Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP). *Games from Within*[blog]. 4 December. Available from: http://gamesfromwithin.com/data-oriented-design [Accessed 16 April 2020].

Longbottom, R. (2017) Multi-Threading Benchmarks [online]. DOI: 10.13140/RG.2.2.10488.75526. Available from: https://www.researchgate.net/publication/320716999_MultiThreading_Benchmarks [Accessed 16 April 2020].

Meijer, L. (2019) On DOTS: C++ & C#. *Unity Blog*[blog]. 26 February. Available from: https://blogs.unity3d.com/2019/02/26/on-dots-c-c/ [Accessed 16 April 2020].

Papadogiannakis, A., Vasiliadis, G., Antoniades, D., Polychronakis, M. and Markatos, E.P. (2012) Improving the Performance of Passive Network Monitoring Applications with Memory Locality Enhancements. *Computer Communications* [online]. 35 (1), pp. 129-140. [Accessed 16 April 2020].

Bibliography

Acton, M. (2008) Three Big Lies. *CellPerformance* [blog]. 14 March. Available from: https://cellperformance.beyond3d.com/articles/2008/03/three-big-lies.html [Accessed 16 April 2020].

Collin, D. (2010) Introduction to Data Oriented Design. *Dice Coder's Day.* Available from: https://www.slideshare.net/DICEStudio/introduction-to-data-oriented-design [Accessed 16 April 2020].

Hwang, E., Kim, H., Nam, B. and Choi, Y. (2018) Cava: Exploring Memory Locality For Big Data Analytics in Virtualized Clusters. *2018 18th Ieee/acm International Symposium on Cluster, Cloud and Grid Computing (Ccgrid)* [online]., pp. 21-30. [Accessed 16 April 2020].

Kochura, Y., Stirenko, S., Alienin, O., Novotarskiy, M. and Gordienko, Y. (2017) Performance Analysis of Open Source Machine Learning Frameworks For Various Parameters in Single-threaded and Multi-threaded Modes. *Advances in Intelligent Systems and Computing* [online]. 689 (2), pp. 243-256. [Accessed 16 April 2020].

Torp, J. (2010) A Step Towards Data Orientation. *Dice Coder's Day*. Available from: https://www.slideshare.net/DICEStudio/a-step-towards-data-orientation [Accessed 16 April 2020].

Appendix A: Project Log

Date of session / log update	In progress tasks up to this date	Summary of achieved outcomes, be it research or practical	Questions or tasks to be taken forwards
9/24/2019	• Refine proposal for a live memory management system.	• Do more research into low level memory management. • Use DoD but have a wrapper for it so that people using an OOP approach can use it.	• N/A
10/1/2019	• Amendments to certain sections of the proposal with less focus on first person.	• Final idea has been solidified, Guidance on parts such as the time management has been advised on.	• Finish the proposal for the 10th. • Begin creating a basic memory allocator in C++.
10/8/2019	• Finalise proposal for hand-in on the 10th.	• Proofread of the current proposal draft. Given a list of ways to improve it .	• Finish the proposal for the 10th. • Begin creating a basic memory allocator in C++.
10/25/2019	• Begin creating a Memory Allocator in C++.	• Created a basic link list allocator using next fit search. • Set up a small test case to see that the allocator works.	• Extend the allocator to use best fit to get the best sized block for the piece of data being allocated.
11/1/2019	• Implement Best fit search into the memory allocator for memory efficient usage of free allocations.	• Implemented a basic linked list memory allocator using best fit search.	• Start to implement Job system with a thread pooler that uses a metric to measure how many threads to pool.
11/7/2019	• Implement the Job system with a thread pool in order the thread tasks for better performance.	• A wall was hit today realising that my memory manager isn't set up correctly to be able to handle setting up a job system in its current state. Need to discuss with supervisor.	• Implemented Job system with thread pooling. • Work out solution to Memory Manager issue. • Extend memory allocator to include memory block splitting and coalescing.

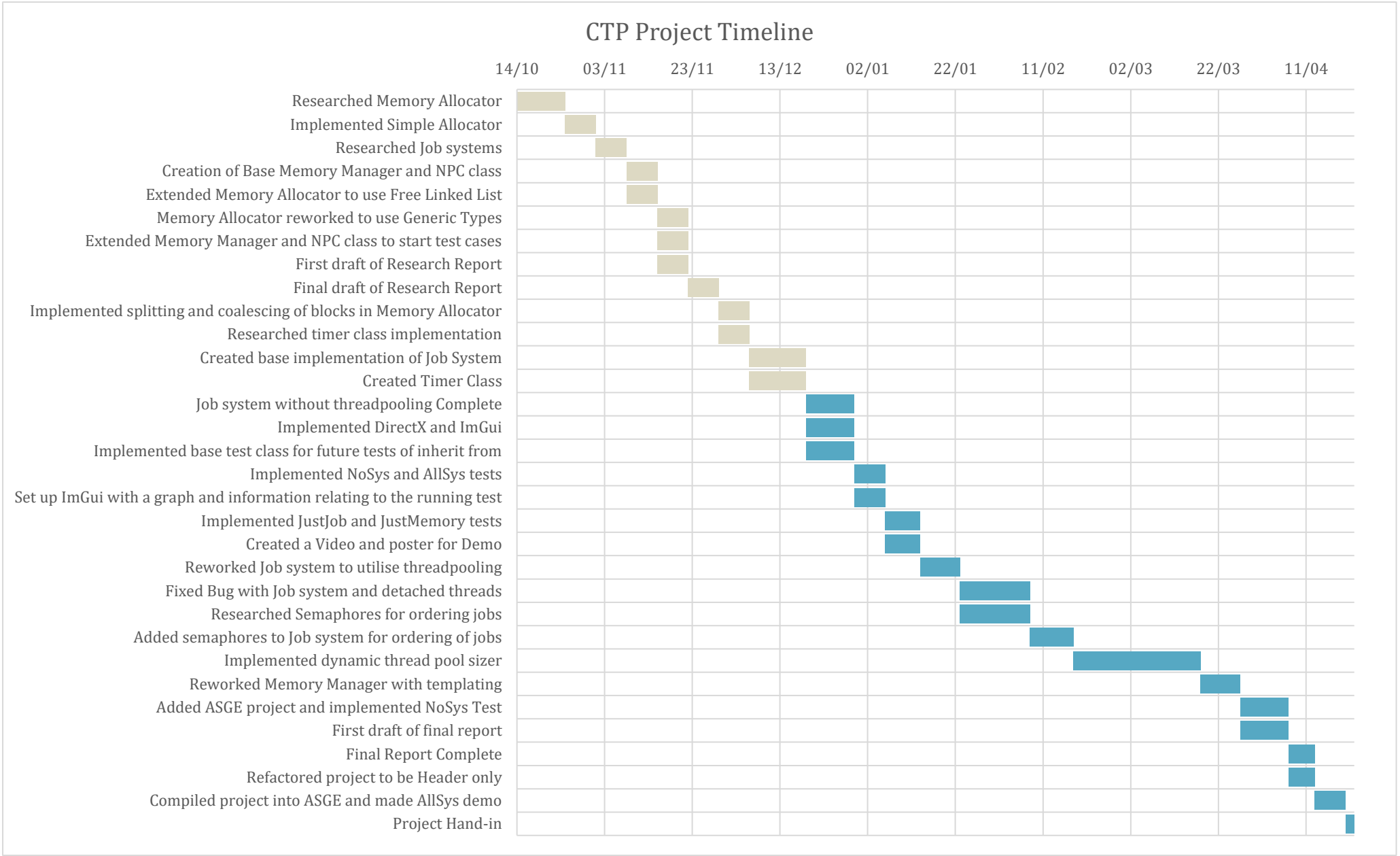
11/8/2019	• Rework Memory Manager to suit the new proposed NPC layout. • Implement Job system utilising the new Memory Manager. • Extend memory allocator to include block splitting and coalescing.	• Extended the Memory allocator to use a more efficient linked list. • Started boilerplate for Job system but behind schedule on that due to issues with the Memory Manager.	• Rework Memory Manager. • Create NPC and NPC manager. • After testing new Memory Manager with NPC test scenario, begin fleshing out Job System. • Splitting and coalescing of memory blocks.
11/15/2019	• Rework Memory Manager. • Create NPC and NPC manager.	• Fixed issue with pointer arithmetic in memory allocator. • Reworked memory allocator for use of generic types. • Created NPC and NPC manager and used Memory manager to store their data in a small testcase.	• Splitting and coalescing of memory blocks now that memory allocator is fixed. • Create Job System. • Write the first draft of the research report.
11/22/2019	• Research and write the first draft of the research report.	• Researched and written the first draft; spoken to supervisor and obtained feedback to further the report.	• Write final draft of research report using the feedback from supervisor. • Splitting and coalescing of memory blocks. • Create Job System.
11/29/2019	• Write final draft of research report.	• Final draft is completed. After speaking to supervisor no more amendments need to be made. • Updated electronic log from my paper notes of supervisor meetings.	• Splitting and coalescing of memory blocks. Create Job system. • Create timer class for getting profiling data.
12/6/2019	• Splitting and coalescing of memory blocks. • Create Job System. • Create timer class for getting profiling data.	• Splitting and coalescing of memory blocks is now complete. • Researched creating a timer class using STD::Chronos to get high precision timestamps between two predefined points in code.	• Create Job system. • Create timer class based of research.
12/19/2019	• Create a Job System. • Create a timer class.	• Created the base structure for the Job system, although limited testing has been completed. • Timer class has been implemented and works based around scope lifetime (When it falls out of scope, invokes call-back).	• Do more robust testing of the Job system. • Begin making all the test cases NoSys, AllSys, JustMem, JustJob. • Implement ImGui for visualisation.

12/30/2019	• Robust testing of Job System. • Implement Test Cases. • Implement ImGui	• Job system works under stress tests although this iteration doesn't current have thread pooling. • Set up a base class for test cases to inherit from. • Implemented ImGui into the project, Demo is working.	• Expand Job System to include Thread Pooling. • Create test cases. • Set up visualisation using ImGui.
1/6/2020	• Expand Job system to include thread pooling. • Create test cases. • Set up ImGui Visualisation.	• Implemented AllSys and NoSys test, Shows promising results when comparing. • Set up a line graph using ImGui with tweakable parameters to adjust test and its iterations.	• Finish the final two tests JustMemory and JustJob. • Polish up the visual in preparation for the demo. • Create Demo video and poster for hand in.
1/12/2020	• Finish the final two tests. • Polish up visuals. • Create Demo video and poster.	• Final two tests (JustJob and JustMem) are implemented with a suite of comparisons for the video. • Adjusted the graph and parameters slightly to be more legible in prep for the Demo video.	• Create the Demo video and poster.
1/14/2020	• Create the Demo video and Poster	• Video has been created using various benchmarking clips and been edited together using keyframe animations in Da Vinci Resolve. • Put together a poster to go with the hand in.	• Expand the Job system with Thread Pooling. • Add system benchmarking to determine how many threads are used in the thread pool.
1/23/2020	• Expand Job system to include thread pooling. • Add system benchmarking to determine thread pool size.	• Reworked Job system to detach threads instead of creating and joining. • Threads now idle if there's no work to process or get given jobs if there are.	• Add system benchmarking to determine thread pool size. • Rework the Memory Manager to make it simpler and easier to use through templating. • Use semaphores to allow for ordering of specific jobs.
2/8/2020	• Add system benchmarking to determine thread pool size. • Rework memory manager to make it easier to use through templating. • Use semaphores to allow for ordering of specific jobs.	• Fixed a bug with the Job system not properly terminating detached threads. Done by utilising STD::Promise and STD::Future.	• Add system benchmarking to determine thread pool size. • Rework memory manager to make it easier to use through templating. • Use semaphores to allow for ordering of specific jobs.

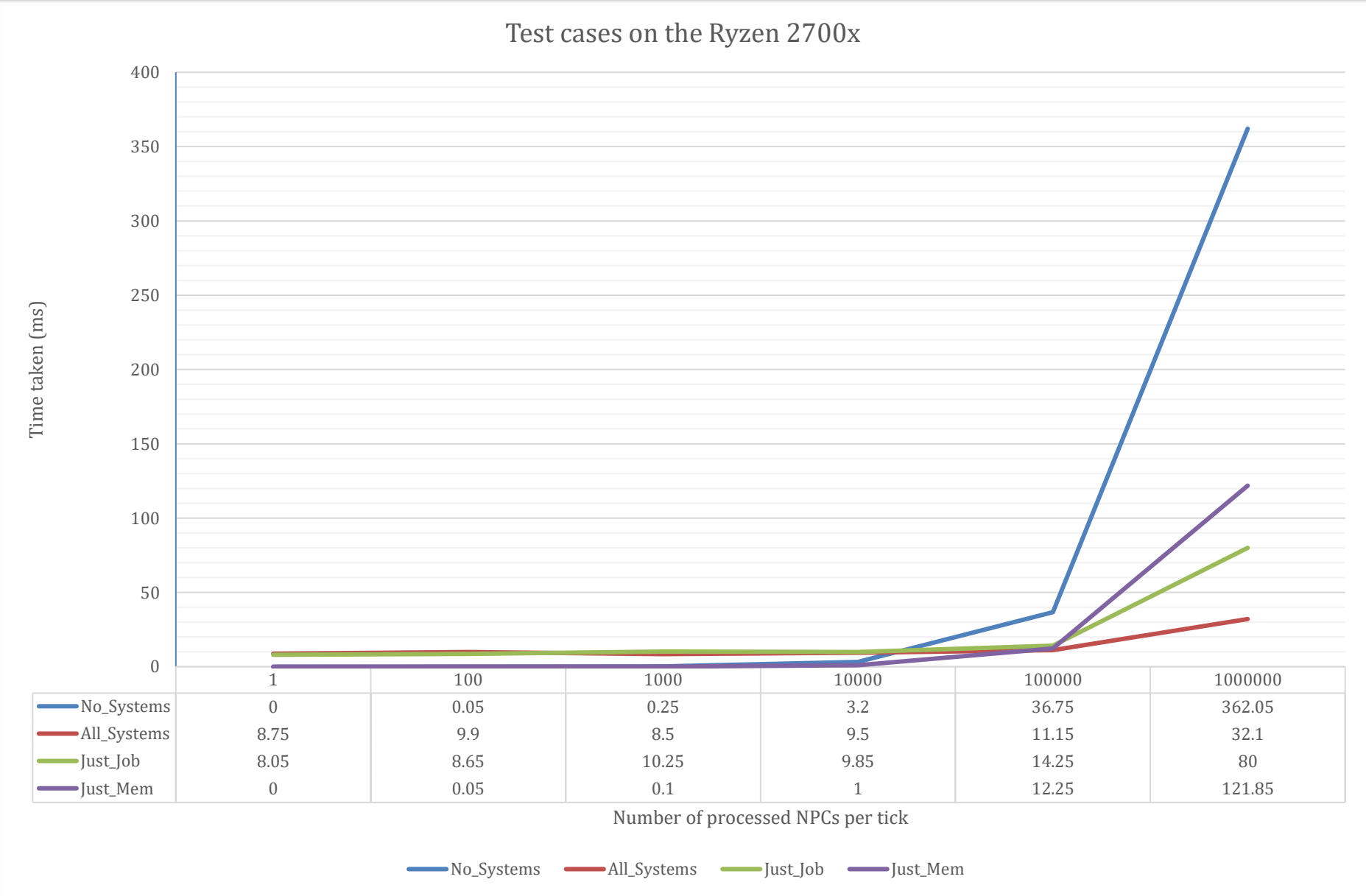
2/18/2020	• Add system benchmarking to determine thread pool size. • Rework memory manager to make it easier to use through templating. • Use semaphores to allow for ordering of specific jobs.	• Implemented the system which allows the user to specify an order on jobs if certain jobs depend on others being completed before it.	• Add system benchmarking to determine thread pool size. • Rework memory manager to make it easier to use through templating. • Create a more game like demo that makes use of each of the systems.
3/6/2020	• Add system benchmarking to determine thread pool size. • Rework memory manager to make it easier to use through templating. • Create a more game like demo that makes use of the systems.	• Spoke to supervisor this week about current progress and why the projects slowed down a little bit. Within the next week a bulk of other modules work should be done so focus can be returned to this.	• Add system benchmarking to determine thread pool size. • Rework memory manager to make it easier to use through templating. • Create a more game like demo that makes use of each of the systems.
3/18/2020	• Add system benchmarking to determine thread pool size. • Rework memory manager to make it easier to use through templating. • Create a more game like demo that makes use of the systems.	• Implemented the system which initialises the thread pool size based upon the hosts system. • Test on both my Desktop and Laptop displaying results that are slightly above base concurrent thread amount which is good as any more than that and it starts to lock up the computer.	• Rework Memory Manager to make it easier to use through templating. • Create a more game like demo that makes use of the systems. • Bullet point the final report writeup.
3/27/2020	• Rework Memory Manager to make it easier to use through templating. • Create a more game like demo that makes use of the systems. • Bullet point the final report writeup.	• Memory Manager has now been reworked and is significantly more usable than before. • Bullet points in the final report have been completed.	• Write the first draft of the final report. • Create a more game like demo that makes use of the systems.
4/7/2020	• Write the first draft of the final report. • Create a more game like demo that makes use of the systems.	• Added an ASGE blank project to the repo and set up the basic demo; which is single threaded and using default allocations. • Completed a first draft of the report.	• Finalise the dissertation report. • Rework the systems to Header only ready for implementation in ASGE. • Implement the last part of the demo utilising the imported systems.

4/13/2020	• Finalise the dissertation report. • Rework the systems to header only ready for implementation in ASGE. • Implement the last part of the demo utilising the imported systems.	• Final writeup of the dissertation report is complete. • Systems have been reworked to be header only. • Systems have been implemented and compiled into the ASGE project.	• Implement the final part of the demo utilising the imported systems. • Final touches before hand-in.
4/20/2020	• Implement the final part of the demo utilising the imported systems. • Final touches before hand-in	• ASGE Demo project is now complete with two test cases, No_systems and All_systems with an fps counter for each comparison. • Project is branched out and tagged Final for hand in.	• N/A

Appendix B: Project Timeline



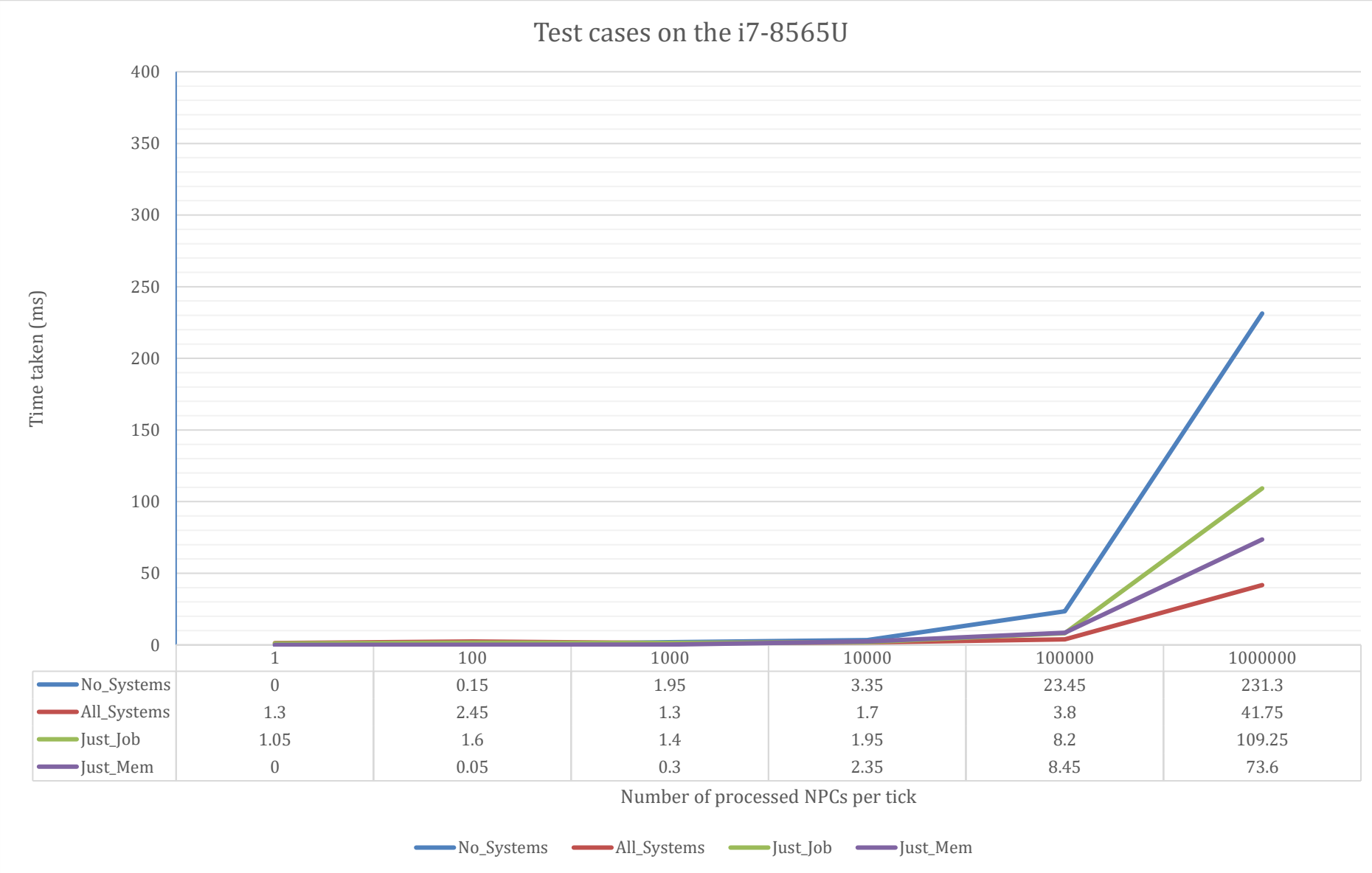
Appendix C: Test results on the Ryzen 2700x:



	1	100	1000	10000	100000	1000000
No_Systems	0	0.05	0.25	3.2	36.75	362.05
All_Systems	8.75	9.9	8.5	9.5	11.15	32.1
Just_Job	8.05	8.65	10.25	9.85	14.25	80
Just_Mem	0	0.05	0.1	1	12.25	121.85

16015338

Appendix D: Test results on the i7-8565U:



	1	100	1000	10000	100000	1000000
No_Systems	0	0.15	1.95	3.35	23.45	231.3
All_Systems	1.3	2.45	1.3	1.7	3.8	41.75
Just_Job	1.05	1.6	1.4	1.95	8.2	109.25
Just_Mem	0	0.05	0.3	2.35	8.45	73.6

16015338

16015338