A Resource Allocation Algorithm for a History-Aware Frame Graph

Roman Sandu
Phystech School of Applied Mathematics and Computer Science
Moscow Institute of Physics and Technology
Institutskiy Pereulok, 9
Dolgoprudny, Moscow Oblast, 141701, Russia
sandu.ra@phystech.edu

Alexandr Shcherbakov
Faculty of Computational Mathematics and Cybernetics
Lomonosov Moscow State University
Moscow, 119991, Russia
alex.shcherbakov@graphics.cs.msu.ru

ABSTRACT

We consider the problem of memory consumption by a real-time GPU-accelerated graphical application. A history of a resource is defined for a particular frame to be the final contents of such a resource at the end of the previous frame. When organizing a graphical application using a frame rendering graph approach, it makes sense to implement automatic serving of resource history read requests of nodes. In absence of history resource requests, allocating resources for a fixed frame graph is the classic problem of *dynamic storage allocation* (DSA). In this paper, we formulate a generalization of DSA that enables memory reuse for resources with history requests and provide a practical approximate algorithm for solving it.

Keywords
frame graph, dynamic storage allocation, resource aliasing, gpu memory reuse, dx12, vulkan

1 INTRODUCTION

Memory Reuse

A computation graph based approach, which has been presented in previous works [ODo17; Wih19], can be considered the current state-of-the-art in real-time render engine design. A typical implementation receives a user-defined directed acyclic graph $(\mathbf{V},\mathbf{E})$, a set of resources $\mathbf{R}$, and a resource usage function $\mathbf{U} : \mathbf{V} \to 2^{\mathbf{R}}$, such that $\mathbf{R} = \bigcup_{v \in \mathbf{V}} \mathbf{U}(v)$. We refer to the tuple $(\mathbf{V},\mathbf{E},\mathbf{R},\mathbf{U})$ as a *frame rendering graph*, or simply a *frame graph*. Vertices in $\mathbf{V}$ are referred to as nodes and represent tasks that dispatch GPU work. Elements of $\mathbf{R}$ represent *transient* resources, temporary per-frame data like the g-buffer, intermediate images during a blur, etc. A node $v \in \mathbf{V}$ may only use GPU resources from the set $\mathbf{U}(v)$ during the dispatched work. The condition that $\mathbf{R}$ is equal to the union across the image of $\mathbf{U}$ ensures that every resource is used at least once. The implementation then executes the nodes every application frame, respecting the dependency order $\mathbf{E}$, and allocates and providing the requested resources to nodes. One of the resources in $\mathbf{R}$ is considered to be the final frame image, which is presented on the screen after the frame graph finishes executing. In some cases, several resources are considered to be the final result of a frame, such as when CPU read-back is needed or when rendering in stereo for a VR application.

This approach has been shown to offer significant advantages. Leaving aside numerous architectural benefits, we focus on memory reuse. Prior to the availability of modern low-level graphics APIs such as Vulkan Direct3D 12, applications usually had to resort to resource pooling when memory consumption of transient resources became problematic. However, a pooling approach has limited memory reuse capabilities due to the abundance of incompatible resource types in graphics programming. In the simplest approach, even two textures with different resolutions cannot be substituted during execution, even though their lifetimes may be disjoint. While certain engineering tricks may be employed to enable greater resource reuse, they are often error-prone, difficult to implement and may incur a runtime performance penalty. Another memory reuse strategy is to create and destroying GPU resources ondemand, although this approach has also proven to be inefficient due to various driver and allocator induced overheads. In presence of a modern graphics API and a frame graph, however, an application runtime is able to take a more nuanced approach to memory reuse.

Related Works

Let $\{v_i\}_{i=0}^{n} = \mathbf{V}$ be the node execution order chosen by the runtime and $\{\rho_i\} = \mathbf{R}$ an arbitrary indexing of resources. For a resource ρ_i, define $l_i = \min_{\rho_i \in \mathbf{U}(v_j)} j$, $r_i = 1 + \max_{\rho_i \in \mathbf{U}(v_j)} j$ the *lifetime* of this resource, and s_i, its size in bytes. With a modern graphics API, we can allocate memory heaps and place resources in them at certain offsets, which are to be chosen such that no two resources overlap in time and memory simultaneously. In other words, memory reuse within such a system reduces to the classic problem of *dynamic storage allocation* [GJ90, p. 226] (DSA), which in general is stated as follows. For an arbitrary set of allocations (l_i, r_i, s_i), find an allocation function $\alpha(i)$ that assigns an offset in memory to every resource, with the minimal value of *makespan* $= \max_i \alpha(i) + s_i$, subject to the following restriction: for any $i \neq j$, either $[l_i, r_i) \cap [l_j, r_j) = \emptyset$, or $[\alpha(i), \alpha(i) + s_i) \cap [\alpha(j), \alpha(j) + s_j) = \emptyset$. Geometrically speaking, given a set of axis-aligned rectangles on a plane, minimize the used vertical sapace by only moving the rectangles along the vertical axis, such that no two rectangles intersect. See figure 1 for an example of gathering lifetime information and building an allocation schedule for a frame graph.

The DSA problem has been relatively well studied over the years [Wil+95; Kie88; Kie91; Ger99; Buc+03]. In the special case of all resources having unit size, the problem trivially reduces to interval graph coloring and is solved in polynomial time by a greedy online algorithm. However, even the case of 2 different sizes is NP-hard[1], so approximate algorithms must be used. Most research focuses on a special case of this problem, called online DSA, where an algorithm must make a decision on $\alpha(i)$ only based on resources $1..i$. This special case formalizes the well-known notion of an allocator inside a language runtime, but it has been proven that there is a lower bound on the efficiency of such an algorithm. Define the load at time t for an instance of DSA to be $L(t) = \sum_{t \in [l_i, r_i)} s_i$, the total load as $LOAD = \max_t L(t)$, OPT to be the optimal *makespan* for that instance, and s_{max}, s_{min} to be the largest and smallest resource sizes respectively. A well-known result [Rob71] of Robson shows that $OPT/LOAD \geqslant 4/13 \cdot (2 + \log_2 s_{max}/s_{min})$. In the context of computer graphics, the s_{max}/s_{min} ratio has been observed by the authors to routinely reach values above 32, which makes the bound be $OPT/LOAD \geqslant 2$. Furthermore, even in the unit-sized allocations case, $OPT/LOAD$ is bounded below by 3 [CŚ88], suggesting that the general case lower bound of Robson may be improved. These results are also known as the fragmentation problem. However, when considering a render graph based system, we are not, in fact, limited to online algorithms. The study of offline DSA started off with reductions to interval graph coloring [CŚ88], the simplest of which has been proven to have a *performance ratio*, the upper bound on *makespan*/*LOAD*, of 80 [Kie88]. After a more advanced reduction scheme [Kie91] was used to achieve a ratio of 6, two consequent results by Gergov decreased the best known ratio to 5 [Ger96] and then 3 [Ger99]. A 2003 paper [Buc+03] then presents an algorithm with a performance ratio of $2 + \varepsilon$, which is the best currently known result. Moreover, that paper presents a polynomial time approximation scheme (an algorithm with a ratio of $1 + \varepsilon$) for the special case of s_{max} being bounded. This result is of particular interest to render graph systems, as resource sizes usually do not exceed a bound induced by the user's display resolution.

Our Contribution

Although for a simple frame rendering graph runtime, the memory reuse problem reduces to DSA, while implementing such a system for Gaijin's Dagor Engine, we have identified a need for a generalization. Many modern computer graphics algorithms use the notion of a *resource history*, the data of a particular resource as it was at the end of a previous frame. Such algorithms, among others, include various screen-space effects like ambient occlusion [Jim+16] and reflections [Sta15], temporal anti-aliasing [YLS20], and occlusion culling techniques [Jim+16]. In fact, among the resources currently tracked through the frame graph in Dagor Engine, almost half require the history to be read at some point while computing a frame. It is clear to see that the previously described mathematical model does not permit tracking such resources inside a frame graph and reusing their memory while the resource is not needed by any node.

Our contribution is as follows. First, we generalize the DSA problem to cyclic time, repeating the same resource usage schedule across two frames. Second, we present a best-fit greedy algorithm that solves this generalization in $O(n \log n)$ time with good practical performance ratio. Finally, we show that no algorithm of a certain natural class, similar to the classic interval graph coloring algorithm, can guarantee an optimal solution for the case of unit-sized resources.

2 RESOURCE HISTORY REQUESTS

Problem statement

We extend the $(\mathbf{V}, \mathbf{E}, \mathbf{R}, \mathbf{U})$ frame graph formalization introduced earlier with the addition of resource history

[1] Although unpublished, this result is due to Stockmeyer, 1976, according to [Buc+03].

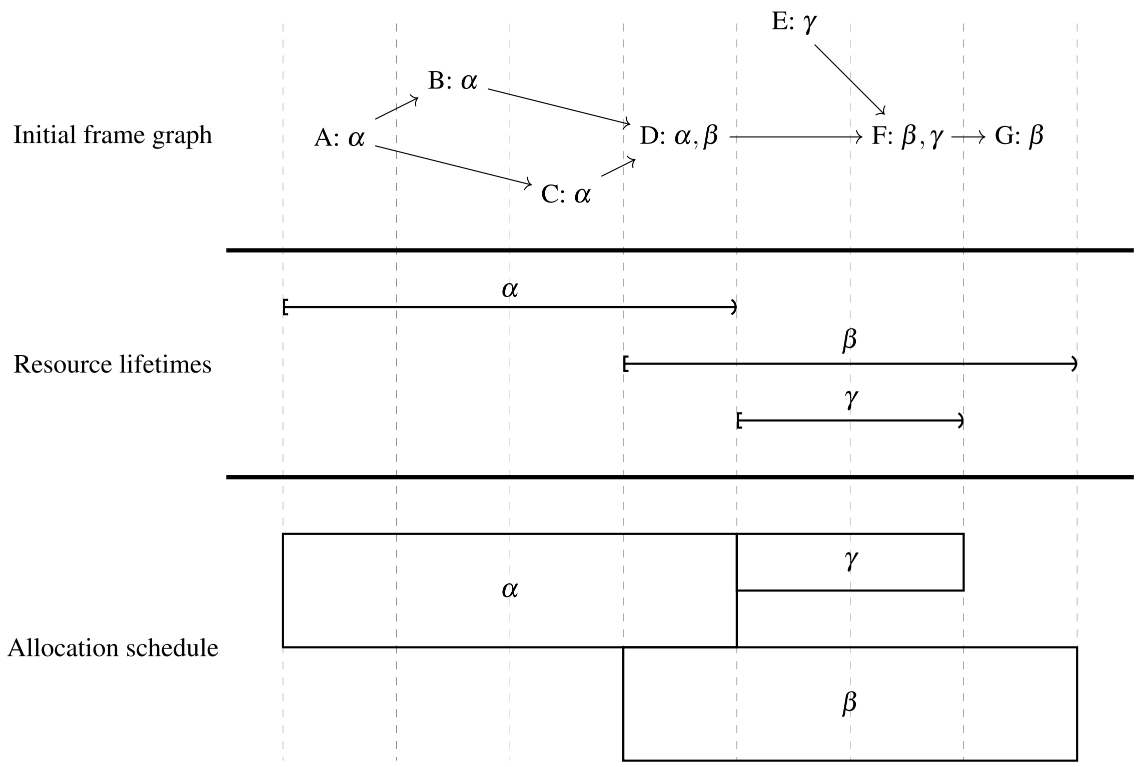

Figure 1: A visualization of a sample frame graph compilation process. Nodes from A to G list the used resources after a colon. Nodes are executed from left to right, and the horizontal axis represents time common to all 3 subfigures. Vertical guidelines represent moments between node executions, when resources start or end their lifetimes. The middle subfigure shows segments that denote lifetimes of resources α, β and γ. Finally, on the bottom, an example allocation schedule for these resources is shown, where the vertical axis represents memory locations. This example can have the following interpretation. α is the g-buffer of an application, β is the final picture, while γ is a low resolution temporary image for particles. Nodes A through D clear, draw things to and resolve the g-buffer into γ respectively, node E renders particles into γ, node F blends γ into the final picture, and G applies tone-mapping to β.

usage function $\mathbf{H} : \mathbf{V} \to 2^{\mathbf{R}}$. With this addition, we will need to start differentiating between the *logical* resources in $\mathbf{R}$ and underlying physical GPU resources. Nodes that read resource history usually use it to produce the same logical resource for the current frame. As such, we create two physical resources for every logical resource and alternate between them on even and odd frames. Note that the sizes of the two physical resources are the same and are determined by the logical resource. We represent a lifetime of a physical resource as an arbitrary pair of elements of $\mathbb{Z}_{2n}$, the cyclic group of order $2n$, and build a resource (de)allocation schedule over two consecutive frames (recall that n is the number of nodes in the graph). Somewhat abusing the notation, we denote such pseudo-intervals in $\mathbb{Z}_{2n}$ as $[l,r)$, and for each frame graph resource $\rho_i \in \mathbf{R}$ the two corresponding physical resources are $[l_i^e, r_i^e)$ and $[l_i^o, r_i^o)$. Geometrically, we now need to place the $2|\mathbf{R}|$ axis-aligned squares on an infinite cylinder $\mathbb{Z}_{2n} \times \mathbb{Z}_{\geqslant 0}$ with no intersections, such that the taken vertical (along the infinite axis) space is minimal. Although the previously defined values l_i and r_i are not applicable to our generalization, for brevity, we define the even and odd frame lifetimes in terms of them and a new value $r_i' = 1 + \max_{\rho_i \in \mathbf{H}(v_j)} j$:

$$\begin{aligned}
l_i^e &= l_i, \\
l_i^o &= n + l_i, \\
r_i^e &= \begin{cases} n + r_i', & r_i' \text{ well-defined} \\ r_i, & \text{otherwise} \end{cases} \\
r_i^o &= \begin{cases} r_i', & r_i' \text{ well-defined} \\ n + r_i, & \text{otherwise} \end{cases}
\end{aligned}$$

Note that r_i' is well-defined iff there is at least one history request for i in $\mathbf{H}$. See figure 2 for a visualization of these lifetimes. Finally, we equate any such lifetime pseudo-interval with the set of elements of $\mathbb{Z}_{2n}$ they contain: for $[l,r)$, if $l < r$, the set is $\{l, ..., r-1\}$, and $\{0, ..., r-1\} \cup \{l, ..., 2n-1\}$ otherwise. Note that for a pair $[x,x)$, the associated set is the entirety of $\mathbb{Z}_{2n}$. All regular set operations apply.

We now are ready to state our generalization of DSA, *cyclic dynamic storage allocation* (CDSA). Given a

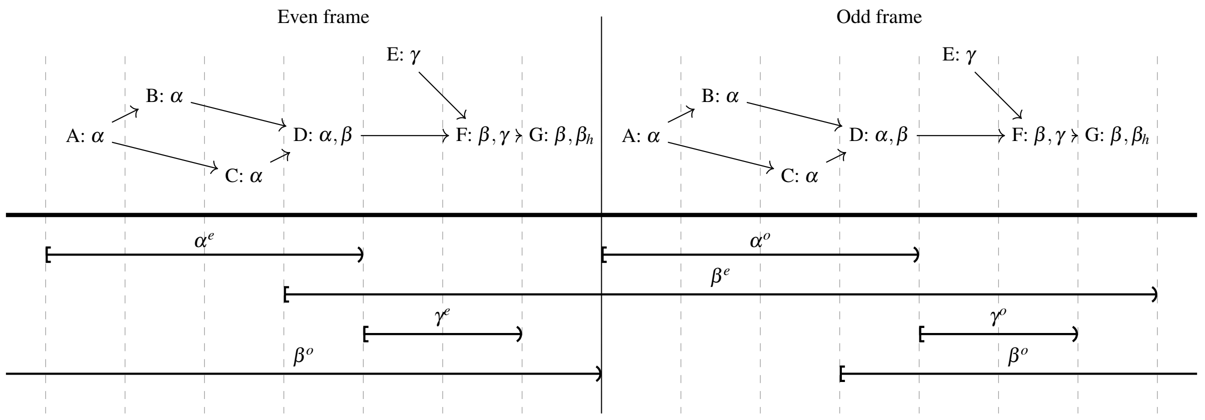

Figure 2: Visualization of physical resource lifetimes for two consequent frames, notation analogous to figure 1. Here, β_h represents a logical resource history read request of a node, while greek letters superscripted by e and o represent physical resources produced from corresponding logical resources on even and odd frames, respectively.

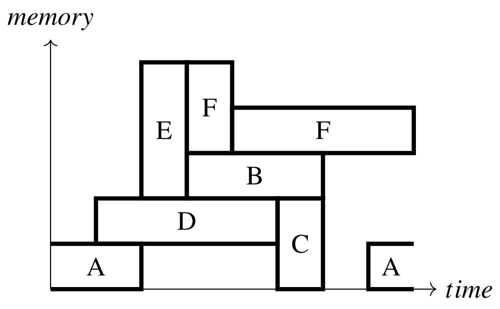

Figure 3: An optimal packing for an instance of CDSA. Here, $OPT = 5$, while $LOAD = 4$. Note that resource A has length 3 and wraps around to zero at the end of the timeline.

timeline size $T \in \mathbb{N}$, a set of arbitrary pairs of elements of $\mathbb{Z}_T$, called allocations, denoted as $\{[l_i, r_i)\}_{i=1}^m$ and each equipped with an integral size $s_i > 0$, find an allocation function $\alpha : \{1, ..., m\} \to \mathbb{N}_{\geqslant 0}$ that minimizes the value $makespan = \max_{1 \leqslant i \leqslant m} \alpha(i) + s_i$, such that for every pair of allocations $i \neq j$ either $[l_i, r_i) \cap [l_j, r_j) = \emptyset$ (where the intersection is interpreted as explained above), or $[\alpha(i), \alpha(i) + s_i) \cap [\alpha(j), \alpha(j) + s_j) = \emptyset$. See figure 3 for an example of a solved CDSA instance.

Before proceeding to our analysis of this problem, let us recall that for DSA, the load at time point t is defined as $L(t) = \sum_{t \in [l_i, r_i)} s_i$, and the total load as $LOAD = \max_{t \in \mathbb{Z}_T} L(t)$. These notions trivially extend to CDSA. Note that for an instance of DSA with unit allocation sizes, henceforth called UDSA for brevity (UCDSA for unit CDSA respectively), the first-fit greedy algorithm proves that $OPT = LOAD$.

Counterexamples

After stating CDSA, a question that arises naturally is whether $OPT = LOAD$ also holds for unit CDSA, and whether first-fit can be extended to find this optimal solution. To try and in some sense answer the second question, consider an arbitrary greedy algorithm of the following form (see algorithm 1).

Algorithm 1 General form of a greedy scanline algorithm for solving UCDSA. Here, "arbitrary" means determined by a concrete algorithm.

```
$X \leftarrow$ the set of allocations
$t \leftarrow 0$
Choose $\alpha$ for all allocations alive at 0 sequentially
        and remove them from $X$
**repeat**
    $i \leftarrow$ element of $X$ with $l_i - t \mod T$ smallest
    Choose $\alpha(i)$ arbitrarily such that no intersections
            with previous allocations occur
    Remove $i$ from $X$
    $t \leftarrow l_i$
**until** no elements remain in $X$
```

The algorithm starts its scanline at time point 0. We argue that this is a reasonable assumption, as any other strategy of picking the starting point can be defeated by adding more resources to a counter-example. All resources alive at the starting time point are allocated sequentially, giving them the smallest α value that does not cause intersections. The algorithm then proceeds to scan the allocations in order of their lifetime start points with respect to the overall starting position and choose α for them using some arbitrary strategy, until no elements remain. Furthermore, we require that this strategy depends only on the lifetime of the current element, as well as the *back profile* and current *front profile*, defined as follows. Consider the start of the algorithm, where it choses the allocation function for all resources alive at 0. For every such resource with $\alpha(i) = s$, the back profile is defined as $p_b(s) = r_i$. For all other values of s, the back profile is defined to be 0. The initial front profile is similarly defined to be $p_f^0(s) = l_i$. After each iteration of the algorithm, a new front profile is defined in terms of the previous one as follows. If the allocation decision made on iteration j is $\alpha(i) = s$, then $p_f^j(s) = r_i$. For all other values of s, leave $p_f^j(s) = p_f^{j-1}(s)$.

Note that this generalized algorithm is applicable to UDSA. In fact, if we restrict it to only UDSA, first-fit becomes an instance of such an algorithm and gives the optimal answer. Now consider two instances of UCDSA shown in figure 4. It is clear that both the

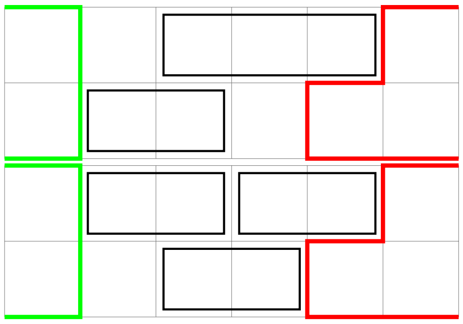
Figure 4: Counter-example to optimality of the generalized greedy UCDSA algorithm. Front profile shown in green, back profile in red, pending allocations in black.

front and back profiles, as well as the chosen resource are equal on the first iteration of the algorithm for the two instances shown. Therefore, the algorithm must choose the same offset for both instances. Obviously, picking the offset not depicted in the figure for either of the instances cannot yield an optimal solution at the end of the algorithm. Moreover, an analogous situation can easily occur in less contrived instances of UCDSA. Therefore, this demonstrates that the naive greedy approach does not in fact solve even UCDSA. Consequently, none of the existing works that reduce DSA to UDSA can be generalized to CDSA without loss of effectiveness.

We now go on to show that in fact even UCDSA is a harder problem than UDSA by proving that $OPT \geqslant 3/2 \cdot LOAD$ for UCDSA. In fact, a stronger assertion can be made: there are infinitely many instances of UCDSA for which the inequality holds. This lower bound obviously holds for general CDSA as well. The proof consists of a single picture, see figure 5. By stack-

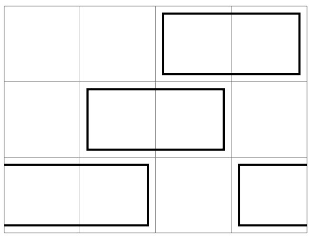
Figure 5: Instance of UCDSA where $OPT = 3$ but $LOAD = 2$

ing the instance in the figure vertically n times, we get an instance of size $3n$ where $LOAD = 2n$ but $OPT = 3n$. The result is evident.

Practical algorithm

We next present a greedy algorithm for CDSA that although has unbounded error in the general case, shows more than acceptable performance in practice. This is because CDSA instances produced by a frame graph often have a very particular structure: most resources are transient textures with sizes that are integral fractions of the user's monitor resolution. The algorithm (2) fol-

Algorithm 2 Proposed best-fit greedy scanline algorithm for solving CDSA

```
$X \leftarrow$ the input set of allocations
$Y \leftarrow \emptyset$ – set of alive allocations
$A \leftarrow \emptyset$ – set of free
        blocks $(offset, size, until)$
$H \leftarrow 0$ – current heap size
$t_0 \leftarrow$ time point with smallest $L(t)$
$t \leftarrow t_0$
for allocation $i$ alive at $t$ do
    $\alpha(i) \leftarrow H$
    $H \leftarrow H + s_i$
    Move $i$ from $X$ into $Y$
end for
repeat
    $i \leftarrow$ element of $X$ with $l_i - t \mod T$ smallest
    Move $i$ from $X$ into $Y$
    for $j \in Y$ do
        if $j$ is no longer alive then
            Remove $j$ from $Y$
            $until \leftarrow r_j$ if $j$ is alive at $t_0$, $\infty$ otherwise
            Add $(\alpha(j), s_j, until)$ to $A$
            Defragment $A$
        end if
    end for
    if picking from $A$ will fail on next step then
        Add a block $(H, s_i, \infty)$ to $A$
        Defragment $A$
        $H \leftarrow H + s_i$
    end if
    $a \leftarrow$ block with smallest $size \geqslant s_i$ in $A$ such
        that $[l_i, r_i) \cap [until, t_0) = \emptyset$ or $until = \infty$
    Remove $a$ from $A$
    $\alpha(i) \leftarrow a.offset$
    if $a.size > s_i$ then
        Add $(a.offset + s_i, a.size - s_i, a.until)$ to $A$
        Defragment $A$
    end if
    $t \leftarrow l_i$
until no elements remain in $X$
```

lows the general greedy scanline form from the counterexamples section, but supports non-unit weights by tracking free allocated blocks. The block to use for an incoming allocation is chosen according to the best-fit strategy. If no such block exists, a new one is allocated at the current highest used offset. When a resource's lifetime ends, we return the block used for it to the free list A. Every time a block gets added to A, we defragment the list by merging blocks adjacent in memory into a single block. The non-trivial idea here is to additionally store an "available until" marker on each free block, that represents the back profile. When picking a free block from the list for an incoming allocation, we ensure that the allocation will not intersect with any of the allocations that were alive at t_0 by rejecting blocks that become unavailable during the current resource's lifetime. It is also important that when defragmenting the free list A, we never merge blocks with unequal $until$ markers. For this to not inhibit all defragmentation, only resources that were alive at t_0 actually store their r_i in $until$. In all other cases the sentinel value ∞ is used, that tells the algorithm that there is no "available until" limit for a block. Note that the value t_0 cannot be used as the sentinel, as it conflicts with the case of a $[t_0, t_0)$ always-alive resource that is created at time point t_0.

Note that this algorithm obviously does not have a bounded performance ratio even for DSA, as a simple sequence of $[i, i+2)$ allocations of size 2^i, $i = 0..n$, will lead to extreme fragmentation and $makespan$ for the result will be $2^{n+1} - 1$, while load is $3 \cdot 2^{n-1}$. However, despite its theoretical limitations, we have observed that the algorithm provides solutions of more than acceptable quality in practice.

3 EXPERIMENTAL RESULTS

We implemented our algorithm in Gaijin's Dagor Engine and ported a significant part of Enlisted's rendering code into a frame graph system. As of May 2023, the frame graph for Enlisted on ultra presets consists of 81 nodes and tracks 27 resources, 14 of which have their history read by at least one node. As one can see from figure 6, a lot of resources alias with resources that cross the frame boundary. This aliasing saves about 10% of memory, or 6 MB, when compared with allocating frame boundary crossing resources separately and never reusing their memory. It must however be noted that a lot of GPU resources are not managed by the render graph system in Enlisted yet, so we expect to see an improvement in these numbers, as suggested by our synthetic tests that follow. The implementation runs in $O(n \log n)$ time, or about 20 µs on Enlisted's frame graph, which potentially enables mid-game changes to the structure of the frame graph.

As our data set for measuring characteristics of the proposed algorithm is extremely limited, consisting of several quality presets in a couple of games, we borrow the bootstrapping technique from statistics. Varying the game and quality presets, we gathered discrete distributions for the following data: resource lifetime length $|[l_i, r_i)|$, resource size s_i, and timeline length T. We assume that lifetime length and resource size are independent random variables, and that a resource is equally likely to begin its lifetime at any time moment. Synthetic tests for N resources were generated by resampling these discrete distribution and choosing l_i uniformly. We then ran the algorithm on these synthetic tests in two configurations, 2000 times each: with and without prohibiting aliasing for resources alive at time point t_0. This simulates a naive treatment of history requests, i.e. allocating resources with such requests separately and never reusing their memory. With this naive treatment, our algorithm becomes a variation of an online greedy allocator, commonly employed by render graph implementations. Our results are presented in figure 7. As can be seen from the plots, the algorithm shows a competitive performance ratio of around 1.1 on average, which is significantly better than what handcrafted counter-examples might suggest. A clear trend can be seen in the plots with history reuse, see figure 8. For inputs distributed similarly to real data produced by a graphics application, the algorithm shows a consistent improvement in memory reuse with increasing resource count, both on average and in the 90th percentile. The same does not hold true for tests with naive treatment of history-requested resources: the ratio instead increases with resource count.

4 CONCLUSION

We've presented a novel memory saving strategy for real-time graphics applications based on a render graph architecture. By treating resources that must outlive a frame boundary due to history read requests uniformly with all other frame graph resources and generalizing the classic DSA problem, we are able to decrease the competitive performance ratio by a significant margin and save about 10% of memory on average. Even though generally speaking CDSA is a harder problem than DSA, as shown by our counter-examples, a greedy approach can still yield good results in practice. We suspect that the presented algorithm has bounded error when restricting it to inputs with bounded allocation size, but are yet to prove this. It must however be noted that in no way can the presented algorithm be considered optimal, even for the use-case of computer graphics. Even small reductions in VRAM usage can have a significant impact on the performance of large-scale high-performance applications, or applications running on memory-constrained devices, such as smartphones, and the greedy nature of this algorithm suggests that further research should be able to find better algorithms to solve CDSA. Of especial interest is generalizing the DSA algorithm from [Buc+03], as it seems to have good performance characteristics on data sets with bounded resource size and highly clustered distribution of resource sizes.

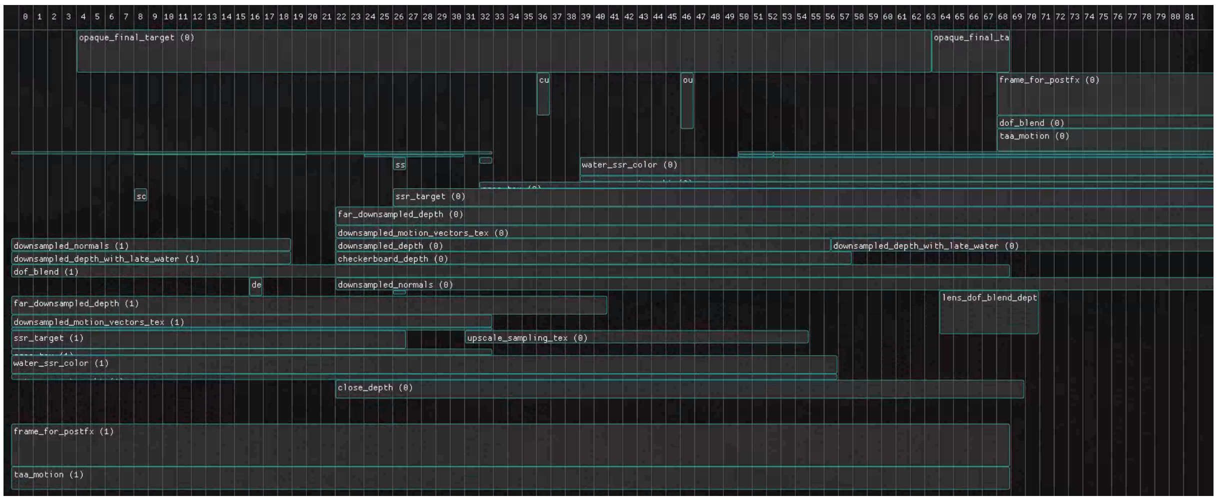

Figure 6: Resource allocation schedule in Enlisted for an even frame on ultra graphics. Horizontal axis is time, vertical axis is memory. Resources with history reads span outside of the 0-81 node range.

Further questions of interest include theoretical properties of the CDSA problem. General algorithms with bounded performance ratio and algorithms for special cases, especially polynomial approximation schemes, are yet to be found. Furthermore, even for the uniform CDSA case, it is not clear whether our lower bound of $OPT/LOAD \geqslant 3/2$ is optimal, and whether an efficient optimal algorithm exists.

5 REFERENCES

[Buc+03] Adam L. Buchsbaum et al. "OPT versus LOAD in dynamic storage allocation". In: *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*. 2003, pp. 556–564.

[CŚ88] Marek Chrobak and Maciej Ślusarek. "On some packing problem related to dynamic storage allocation". In: *RAIRO - Theoretical Informatics and Applications* 22.4 (1988), pp. 487–499.

[Ger96] Jordan Gergov. "Approximation algorithms for dynamic storage allocation". In: *European Symposium on Algorithms*. Springer, 1996, pp. 52–61.

[Ger99] Jordan Gergov. "Algorithms for compile-time memory optimization". In: *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*. 1999, pp. 907–908.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1990.

[Jim+16] Jorge Jiménez et al. "Practical real-time strategies for accurate indirect occlusion". In: *SIGGRAPH 2016 Courses: Physically Based Shading in Theory and Practice* (2016).

[Kie88] H. A. Kierstead. "The Linearity of First-Fit Coloring of Interval Graphs". In: *SIAM Journal on Discrete Mathematics* 1.4 (Nov. 1988), pp. 526–530.

[Kie91] Hal A. Kierstead. "A polynomial time approximation algorithm for dynamic storage allocation". In: *Discrete Mathematics* 88.2 (1991). Publisher: Elsevier, pp. 231–237.

[ODo17] Yuriy O'Donnell. "FrameGraph: Extensible Rendering Architecture in Frostbite". Game Developers Conference. 2017.

[Rob71] John Michael Robson. "An estimate of the store size necessary for dynamic storage allocation". In: *Journal of the ACM (JACM)* 18.3 (1971), pp. 416–423.

[Sta15] Tomasz Stachowiak. "Stochastic Screen-Space Reflections". SIGGRAPH. 2015.

[Wih19] Graham Wihlidal. "Halcyon: Rapid innovation using modern graphics". Reboot Develop. 2019.

[Wil+95] Paul R Wilson et al. "Dynamic storage allocation: A survey and critical review". In: *Memory Management: International Workshop IWMM 95 Kinross, UK, September 27–29, 1995 Proceedings*. Springer. 1995, pp. 1–116.

[YLS20] Lei Yang, Shiqiu Liu, and Marco Salvi. "A survey of temporal antialiasing techniques". In: *Computer graphics forum*. Vol. 39. 2. Wiley Online Library. 2020, pp. 607–621.

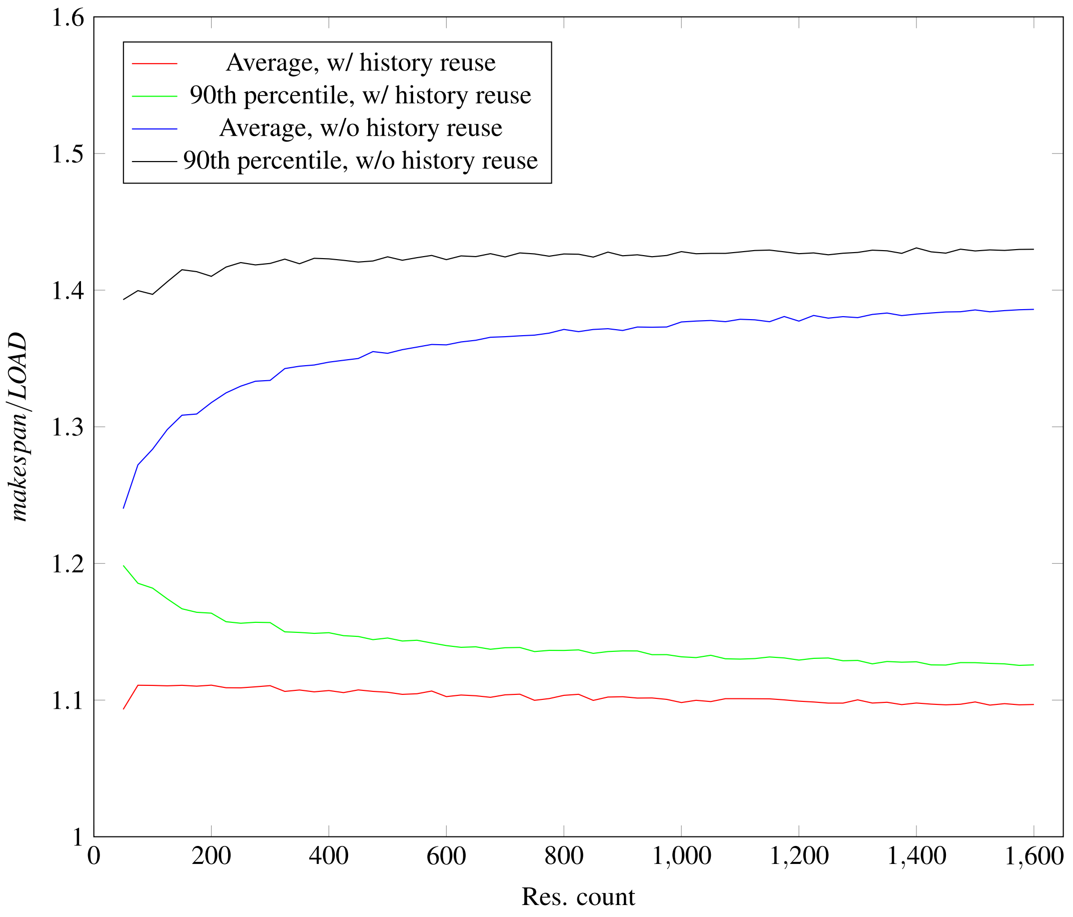

Figure 7: Performance measurements for our best-fit greedy scanline CDSA algorithm. Cumulative over 2000 runs on synthetic tests, bootstrapped from production data.

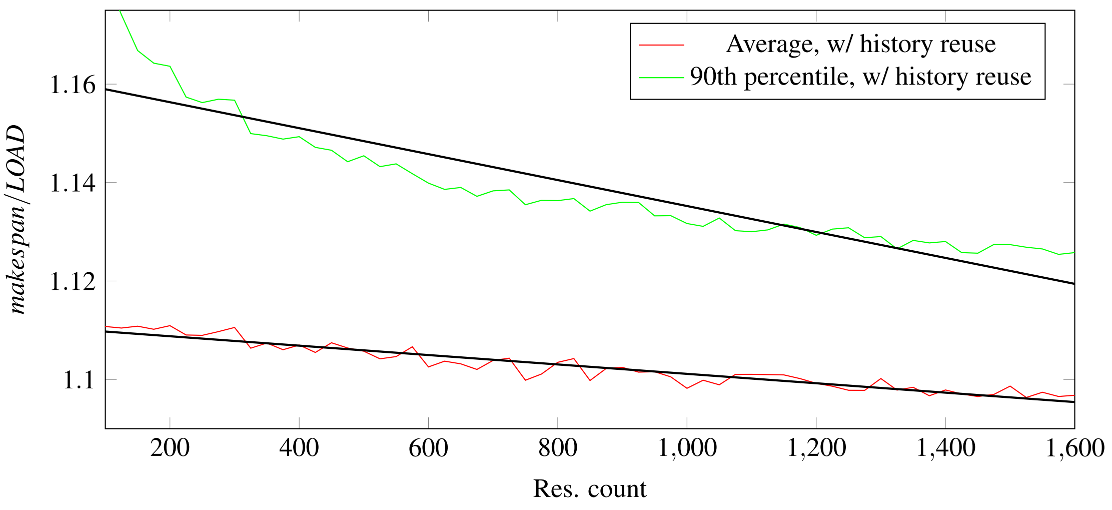

Figure 8: Enlarged plots with history reuse from figure 7. Clear downward trend can be observed.