Exploring Entity-Component-Systems for Cache Optimization: Component Groups & Auto-Grouping

Jay Idema, With Supervisor: Bin Ren

***Index Terms*—ECS, Cache Optimization, Programming Paradigm, Boolean Set Logic, SoA.**

I. Introduction

A. The Problem

IN software development, the use of arrays to store data in contiguous memory is common practice. However, when managing polymorphism (using virtual classes and dynamic dispatch for example), this contiguous approach may not be feasible due to the difference in the memory sizes of classes with shared virtual interfaces. Naively attempting to store different classes with a shared interface as though they all share the same size can result in memory overwrites; one object writes over parts of data for other objects within a contiguous array. A common solution is to use arrays of polymorphic pointers. This can lead to many performance issues: pointer indirection overhead (due to both the obvious pointer indirection and the virtual function lookup table indirection) and no useful memory patterns for cache pre-fetching (due to the non-contiguous storage of the actual class data).

B. The Entity Component System (ECS)

An Entity Component System (ECS) is a software architectural design pattern used primarily in game development, but also in other simulation domains. Its core intention is to separate data into independent, reusable **components** and functionality into independent **systems** which operate on a subset of these components. This separation prioritizes **entity** composition (by component ownership) over explicit inheritance relationships.

At the core of an ECS are three main elements: **entities**, **components**, and **systems**. An entity is an object (composed of components) and is typically represented by an entity handler ID and a list of its component types. A component is a data type that represents specific aspects or properties of an entity. Finally, a system is a set of operations performed on one or more components of an entity.

It is key to note that a system operates on the components of a single entity at a time and does not operate on components for entities which do not meet the component requirements of the system. For example, **System(A, B)** will only operate on components owned by entities with both A and B components); likewise, **(System(A, B, !C)** will only operate on components owned by entities which have both A and B components and do NOT have a C component. It is also common for systems to require that the entities they operate on to have components which are not used by the system.

Components are typically stored in memory via one of two primary ways: by composition Archetypes, or by fully contiguous arrays. *We focus here on an implementation using fully contiguous component arrays.*

Entity-Component-Systems operate well with tick-based simulations. During each tick or 'frame' of the simulation, an ECS may update components using its the systems. Changes to the component composition of entities is typically queued during a tick; with all queued changes finally committed in bulk at the end of a tick.

Most work in ECSs has been performed very recently [1]–[3], however the concept of organizing components into contiguous arrays is very similar to the 'Structure of Arrays' pattern [4], [5] used to better optimize code (inlining of functions and vectorizing operations) and to better optimize memory access patterns over a traditional 'Array of Structures' commonly seen in Object Oriented Programming (see Related Work).

II. Component Groups

To achieve better cache and access pattern performance, it is common to specify **'groups'** of related components. An entity, which has all the components of some group and doesn't have any of the components which a group excludes, is considered a member of that group. Groups can be represented by a boolean expression, such as (A & B), meaning entities with A and B, or (A & !B & C), meaning entities with A and C but not B. Similarly, the components of an entity can also be represented by a boolean expression based on the components which is has and doesn't have. Mathematically, an entity is in a group if its component boolean expression logically implies that group's expression.

The components of entities that are part of a group are partitioned within each of their corresponding component arrays to maintain a contiguous sub-block of components for this group in each array. These sub-blocks of component arrays do not need to be sorted in any specific way, but each sub-block must be ordered such that the order of owning entities for each component sub-blocks in different component arrays are all the same (see Figure 1). A system can then sequentially iterate over the sub-blocks of the component arrays which correspond to a group, rather than needing to use entity handler indirection to find an entity's components.

For systems which iterate over only one component, since all components of a specific type are stored contiguously in memory, cache lines are utilized efficiently and the cache prefetcher has an easy pattern to predict, thus greatly reducing

cache misses and improving cache performance. Additionally, because ECSs can eliminate pointer indirection entirely for eligible systems, compilers can often inline a system's function, then parallelize, unroll, and vectorize a system easily, just as compilers can for 'Structure of Array' data layouts. With properly accommodating groups, both the cache locality and compiler optimization benefits can be extended to systems which iterate over multiple components (or systems which require its entities to have some components and not have others), since each group within the component arrays is partitioned and ordered by entity identically.

A. Overview

The grouping space for a component can be conceptualized as a boolean expression tree with a root equal to the boolean expression of 'has just this component', indicated by the component name henceforth (See Figure 3 for an example). For such a tree, the boolean **OR** of the child expressions of a node is equal to their parent's expression. Suppose that a component array is divided into sub-groups based on the leaf nodes of the tree, in in-order traversal order; this means that a group which is not a leaf still has a contiguous sub-block in the array, since the sub-blocks of two child nodes will be adjacent and thus compose a larger sub-block equivalent to the parent.

For example, for a component A, if we have a group A&B, then the tree has a root node (A) with child nodes (A&B) and (A&!B), and thus the component array A has sub-blocks (A&B) and (A&!B).

This is recursive, so we can further specify the group A&B&C, which results in the node (A&B) having child (A&B&C) and (A&B&!C), and thus the sub-block (A&B) is divided into subsections (A&B&C) and (A&B&!C). The in-order sub-blocks of the component array A therefore are (A&B&C), (A&B&!C), and (A&!B). The group (A&B) still has a contiguous sub-block since (A&B&C) and (A&B&!C) are adjacent and compose all of (A&B).

It is also possible for groups to be stricter than a single added component over a parent node: specifying a group of A & B & C from initially no groups, results in the A component array being sectioned by: (A & B & C) and (A & (!B ‖ !C)) and the B and C component array being similarly sectioned.

B. Entity Management

Of important note, entities can change their components at runtime and components of entities must therefore be added to, moved from, and removed from component sub-blocks at runtime; too much granularity from these group specifications can be detrimental to performance (too large a boolean expression tree for a component degrades the efficiency of maintaining the contiguous nature of component arrays). The typical method for maintaining an array on a removal is shifting the last element of a group to the space where a component was removed, then do the same for the next rightward group until you reach the end of the array [1], [6]. It is typically assumed that component changes are less common than system executions (ticks) for the applications ECSs are designed for, but this does not negate reasonable caution in the efficiency of entity manipulation.

While group sub-blocks do not need to have their components sorted in a specific order, groups in different components with the same boolean expression need to be ordered by entity in the exact same way to gain the maximum benefits of the group; since this allows seamless iteration for a system. This is a key issue with a laissez faire approach to specifying groups; some groups in components may be affected by an entity's addition or removal, while other components are not affected. Component groups will ALWAYS remain grouped in contiguous sub-blocks correctly but are NOT guaranteed to be ordered by entity the same within the same group sub-blocks of other component arrays; this can result in outright incorrect execution if ignored and poor performance if handled using entity handler indirection.

In Figure 1, an entity deletion is handled without a problem, the last component in each sub-block is moved to the start of the sub-block to fill the holes created by Entity 1's removal and the orders of groups (Blue and Yellow) are maintained. However, in Figure 2, the removal of Entity 2 results in mismatched ordering for the red same-expression groups between components A and C.

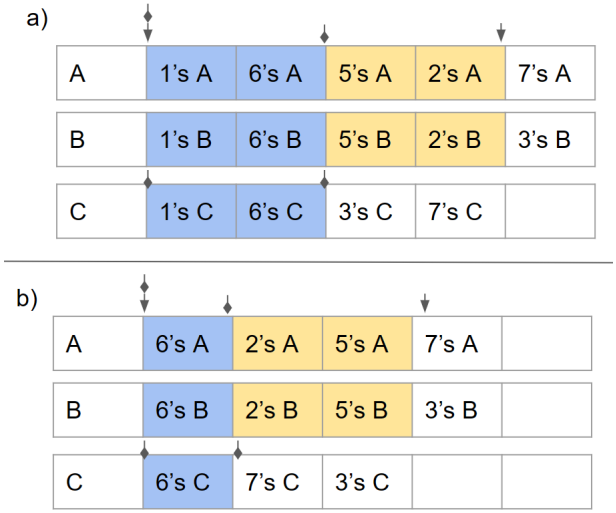

Fig. 1. An entity deletion which maintains all inter-component orderings. Component arrays for components A, B, and C are shown, with explicit groups (A & B & C) (BLUE) and (A & B) (BLUE & YELLOW). Entities are denoted by number, i.e. "1's A" is the A component of Entity 1. 1a shows component arrays A, B, and C before the removal of Entity 1. 1b shows the arrays after 1's removal, note how 2's A and B have been moved from the end of its group to the beginning to fill in the gap made by 1's removal, then 7's A and 3's B are also shifted.

C. Group Logic/Investigation

If we treat the group hierarchies as a set of boolean expression trees, where the left child of a node is sectioned to the 'left' array side of the parent node's array sub-block, then all groups of the tree which are on the 'far' left side of the tree will have their inter-component orderings correctly maintained for ALL components they are a part of. We refer to this subset of groups as 'explicit' groups. If a parent group

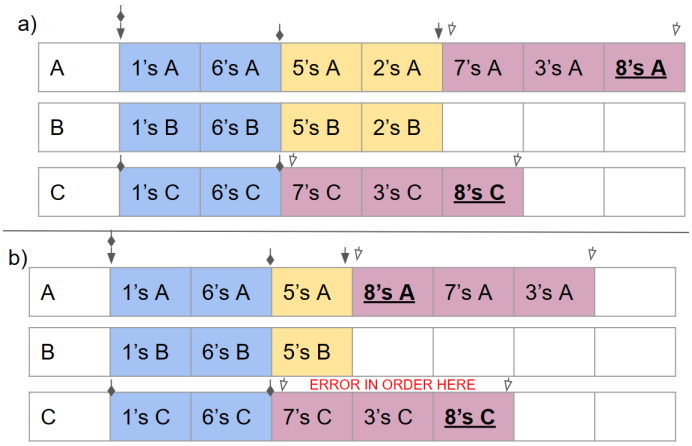

Fig. 2. An entity deletion which does not maintain all inter-component orderings. Component arrays for components A, B, and C are shown, with explicit groups (A & B & C) (BLUE) and (A & B) (BLUE & YELLOW) and (A & !B &C) (RED). 1b shows the arrays before Entity 2 is deleted and 2b shows the arrays after, in a similar manner to the first example. However, since Entity 2 does not have a component C, its removal does not affect the component array for C, only for A and B. This is a problem, since we will need to shift components in the group (A & !B & C) for component array A, but NOT for array C, resulting in a mismatched ordering for the group.

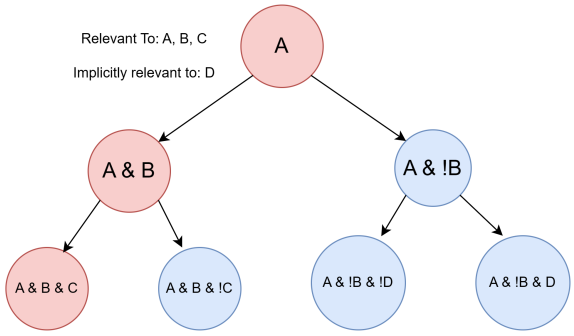

Fig. 3. A boolean expression tree where the set union of a node's children represents itself. The tree also spatially defines how groups will be partitioned in component arrays. Tthe nodes marked in red (leftmost) represent groups that will never be mis-ordered between different component arrays, given the hierarchy of effect which removals and additions have to the tree. If the sub-block within D for A & !B & D were to be added to however, it would not be guaranteed to have its order maintained against A's, since operations affecting A & B might shuffle components in the A & !B & D sub-block without mirroring those changes in D's sub-block for the group.

is explicit, then its left side child will be as well; if a parent group is not explicit, neither child will be. Figure 3 shows a boolean expression tree for components A, B, and C with the groups that are explicit and thus expected to have their order maintained correctly in red for the A, B, and C component arrays; this is not the case for the D component array since it is only included in the non-explicit group **(A & !B & D)** to the far-right.

We analyze mechanics for creating sub-groups which do not violate the well-ordered-by-entity principle, and offer a method for creating component group organizations which maintain the principle. We also offer a simple proof of how and what sub-groups are guaranteed to have their inter-component orders maintained and within what components.

ECSs like EnTT [6] avoid this issue by only allowing a component to be a part of a single fully-ordered group. This is overly conservative but guarantees shared entity order for groups.

As mentioned, other ECSs [7] avoid this issue by implementing a different system for component organization, Archetypes: each entity is typed based on the components it has and doesn't have, and its components are placed in unique component arrays based on their entity's composition type. This is very similar to how a structure of arrays system might need to handle inheritance, adding additional arrays for new fields. It would be extremely beneficial to analyze the performance tradeoffs between our primary focus, the groups strategy, and the Archetype strategy, however the Archetype strategy is likely to be outside the scope of this project.

Our method is furthermore able to synthesize with the Archetypes implementation technique; a single component array can be maintained for all components of entities which match an explicit groups that contains the component, while entity composition types not guaranteed to have shared order are implemented as separate arrays as in the Archetype technique.

III. Groups, Group Trees, and Group Sub-Blocks

A. Boolean Expression Trees and Tree Sets

We define a boolean expression tree (BET) as defined in the introduction, despite this not being a 'common' definition; each node of a tree represents a boolean expression and the boolean OR of a node's two child expressions are equivalent to the expression of that parent. For the purpose of our ECS, Symbols in these BETs represent components.

For any node of any BET, we define a function TrueSymbols(node) which defines the set of 'true' symbols in the node's expression. For example, if Expression($node_a$) = A & !B & C & E, then TrueSymbols($node_a$) = {A, C, E}.

Given the nature of our boolean expression tree, the true symbols of the children of an expression node will always have following relationship:

$$TrueSymbols(node) \supseteq TrueSymbols(node \rightarrow parent)$$

We define the Maintained Symbols set of an entire BET as equal to the True Symbols set of the far left node of the tree. The Maintained Symbols of a tree specify the upper bound of which components MUST use the expressions of the tree as its groups, while maintaining our desired ordering property.

B. User Interface & Immediate Workings

'Raw' boolean expression trees constructed by the user will be turned into groups on a per-component basic. Our ECS maintains multiple disjoint 'raw' BETs, with the condition that the intersection of the Maintained Symbols sets of any two BETs are always empty. Since the components which are affected by all trees are disjoint, we can lower the trees into the components represented by the symbols in their Maintained Symbols set, without fear of BETs conflicting each other.

The nodes of these 'raw' BETs should not be considered component groups quite yet, since these trees are constructed without having a root node which represents a single component, a requirement for a component's group tree. These 'raw', user-defined BETs will eventually be lowered into the components represented by the symbols in their Maintained Symbols set. At that time, the existing BET can be appended to the left of a new root node that simply specifies the component; this maintains BET validity and permits easy lowering to shared-order groups.

In its current implementation, the user specifies groups which are used to construct a set of boolean expression trees which will eventually be lowered into components as groups. Group expressions are always of conjunctive normal form where clauses which are conjoined have no logical ORs, i.e. (A & B), or (C & !D). Group expression specifications are used to create a set of boolean expression trees with the disjoint maintained symbols restriction discussed above, then after all groups have been specified, the trees are used to specify the sub-blocks of each maintained component arrays.

If a user attempts to create a group with expression, expr_g, for which the intersection of its True Symbols with all tree Maintained Symbols is empty for all current trees, then a new tree must be created, since no current tree could possible accommodate it (by the super-set requirement). The group's expression thus becomes the root of this new tree and by the (TrueSymbols(node) $\supseteq$ TrueSymbols(node$\rightarrow$parent)) property stated above, this new tree is guaranteed to explicitly affect and maintain all of the components in TrueSymbols(expr_g).

If a user attempts to create a group with expression, expr_g, for which the intersection of its True Symbols with all current trees' maintained symbols is only NON-empty for a single tree, tree_i, then we can attempt to insert that group expression into that single tree (this is not guaranteed to be possible). We insert nodes by finding the leaf node which has an expression that is implied by the new expression and add a new node as the left child of that leaf node; a right child is also created with an implicit expression (parent & !left). This could result in a node_{new} for which TrueSymbols(node_{new}) == MaintainedSymbols(tree_i); thus all the components in TrueSymbols(node_{new}) will have the shared order for their explicit sub-blocks. It is, however, also possible that TrueSymbols(node_{new}) may have symbols which are not in MaintainedSymbols(tree_i); in this case only the only components which will see maintained order for the group's sub-block are:
TrueSymbols(node_{new}) $\cap$ MaintainedSymbols(tree_i).

Finally, if a user attempts to create a group with expression, $expr_g$ for which the intersections of its True Symbols with all current trees' maintained symbols are NON-empty for multiple trees, then we attempt to insert the expression into all trees with a non-empty intersection, with the additional condition that we do not do insertions that would add symbols to a tree's maintained symbols set. This is equivalent to saying that the expression cannot be explicit. We can do this by adding the expression as the right child, rather than the left child of a leaf node, since the maintained symbols set will never be affected by a right child (non-explicit). This is an overly conservative restriction but ensures our condition that the sets of maintained symbols for all trees are disjoint without rejecting such expressions.

C. Lowering BETs into Component Arrays as Groups

After all groups have been specified, we need to lower the group specifications into each component array individually. We call the special boolean expressions trees used by individual components to organize themselves 'group trees'.

In order to create the groups for the component array, we process each BET one by one. For each BET, we find the explicit (left-side) node closest to the root which has a True Symbols set which contains the component's symbol. This node is affixed to the left side of a node with only the expression ("Component is True"), see Figure 4.

The groups are then ordered in the array as sub-blocks by left-to-right traversal order of the BET. Entities and systems can query for the contiguous sub-blocks relevant to them by traversing the specific group tree for each relevant component.

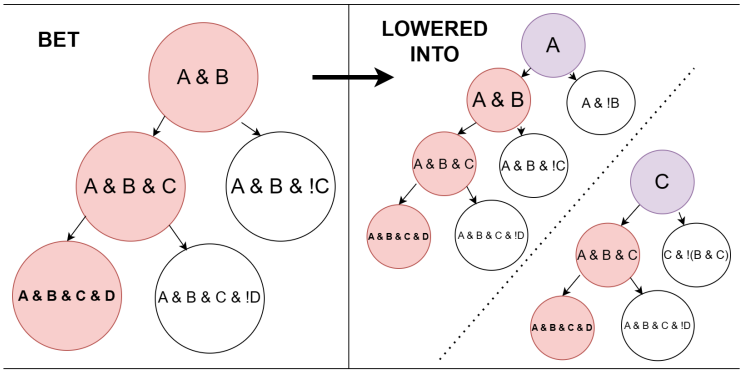

Fig. 4. Left: An example BET created by specifying the expressions (A & B), (A & B & C), and (A & B & C & D). Right: How this BET is lowered into the component arrays of components A and C as group trees. Note that the root of both is *their component* (in purple) and that the left sub-tree of each is the same as the sub-tree of the Left BET rooted at the first left-side node for which its true symbols set contains the component's symbol. Nodes in red are the explicit (left-side) nodes of the tree.

D. How Sub-Blocks are Maintained

As mentioned previously, the leaf nodes of a component's group tree are lowered into the component array by left-to-right traversal order of the tree; each group is allocated a (potentially empty) contiguous region of the array. Components are removed and inserted into groups only at the end of each 'tick'.

Since the start of the array is fixed, changes in group regions to the right of leftward group regions will NEVER affect the ordering or position of those leftward groups; this is a vital property for maintaining order desired entity based orderings (See the Proof section below).

On component insertions into a group sub-block, room is made at the end of the sub-block by recursively pushing the component at the front of rightward sub-blocks to their back, then the new component is inserted into the gap created. On

component removals from a group sub-block, the component is written over with the last component of the sub-block; the gap created is then filled by recursively shifting the last component of rightward groups into the gap. Since component additions and deletions are handled in bulk at the end of every tick, we can make some efficiency optimizations, which still maintain the core contiguous nature of component arrays. This method is amortized to be O(G * M), where G is the number of explicit groups and M is the number of component changes per component.

An array of entity handles is maintained alongside the component array, where each element corresponds to the entity handle of the component at the same index. These are only used in systems which need to utilize indirect handle lookups to get components for the entity. Similarly, a map from an entity handle to the index of a component is maintained.

Since entities will always either have or not have a component, they will always be placed at a leaf group, so it is easy to manage the bounds and organization of the leaf groups. Groups with children, however, are slightly more complicated, but we can take advantage of the fact that a non-leaf node's group is represented by the concatenation of the groups of its two children. We can therefore propagate up the group bounds of a node's children to represent its group, See Figure 5.

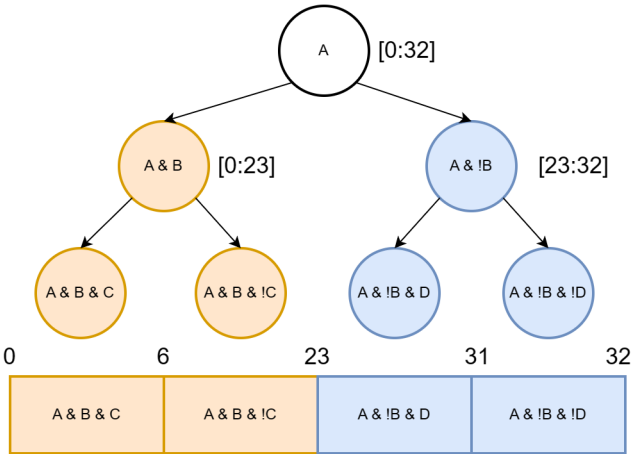

Fig. 5. An example of how a component's array is organized using a group tree, and how contiguous sub-blocks are maintained for nodes of the group tree. The leaves of the tree are made into the lowest granularity sub-blocks in in-order traversal order. Since the sub-blocks for children of a group node are adjacent, its sub-block is represented by the concatenation of both of its children's sub-blocks. For example, the group (A & B) is the sub-blocks from indices [0, 6) and [6, 23), which is indices [0, 23). The orange sub-blocks of the component array for A are guaranteed to have their components share order with components B and C, however the blue sub-block have no such guarantee.

E. Proof Of Maintained Order

We seek to prove that group sub-blocks in different component arrays maintain their order as claimed above.

Given a set of boolean expression trees BETs = $\{B_0, ..., B_n\}$, they are paired with a set of symbols which are 'true' in their far left node, their Maintained Components.

Assume that the intersection of any 2 or more Maintained Components sets for these trees are the empty set; that is:

For $B_i, B_k \in BETs$ (with $i \neq k$),
- (Maintained(B_i) $\cap$ Maintained(B_k)) = $\{\}$.

Given the nature of our boolean expression tree, the True Symbols of a child of an expression node will always be a super-set of that node's true symbols. Assume that we organize groups in a component array using in-order traversal of its generated expression tree, from left to right. Lastly, assume that our ordering algorithm ensures that component additions and removals in any groups only require adjustments from groups to the right of that group (as in our algorithm).

Base Case:
For any $B_i \in BETs$, the far-left node of B_i will never be affected by adjustments of other groups in an array, the group will maintain its ordering across all component arrays in the Maintained(B_i).
Induction:
Since the parent of the far left node is still the farthest left group (it contains the last far left group), it too will not be affected by any groups to the right of it. Because its left child's True Symbols set is a super-set of its True Symbols, any changes to them will affect the components in its true symbol set in exactly the same way. This recursion continues until we reach the root node, which will inherently always be ordered correctly.

IV. Group Tree Traversal and Efficient Tree Navigation

Navigating a boolean expression tree for our purposes (i.e. to determine where to put an entity's components, or what group to attach a system to) would naively require that we use an SAT solver for boolean expressions. For example, we can determine if an entity is contained within a group if a boolean expression, expr_{ent}, representing the components that the entity has and doesn't haven't, implies the boolean expression for the group, expr_{group} (that is we can use an SAT solver to check that $(\neg expr_{group} \wedge expr_{ent})$ has no solution). We can them recursively descend from the root to reach the leaf group node that the entity belongs to. However, SAT solver are inefficient, overly general for our purposes, and do not port well.

Instead, we take advantage of the fact that the boolean expressions of both entities, systems, and explicitly-requested groups will only ever be of a conjunctive normal form which does not use logical ORs.

Using this property, we can navigate the boolean tree using only bit operations. Groups which are user-requested have their expressions represented by 2 bit arrays, one in which a bit is 1 iff its corresponding component is a symbol in the expression (whether positive or negative), and another bit array in which a bit is 1 iff its entities in the group must have the component corresponding to that bit or 0 iff entities in the group must NOT have the component. For this purpose, each component is assigned a UNIQUE bit ID in [0, i), where i is the number of components; these IDs can be used to index a bit array for a particular bit corresponding to a component.

More formal definitions for the representations of entities and systems can be found in the next two section; both have

a similar bit-array-based representation for their component representation. Using these bit-arrays, we can perform very fast bit-wise operations to determine if a group's expression is equivalent to a system's expression, or if an entity's composition implies a group.

It is vital to note that we can only represent explicitly-requested groups as these useful, compact bit arrays. Say for instance that a user requests (A&B&C) as the only group, then component A will have a group tree with leaf nodes, (A&B&C) and $((A\&(!B\|!C))$; and $(A\&(!B\|!C))$ is not represent-able with 2 bit arrays.

However, since at least one child of every node (by our method of insertion) is of the proper form (and therefore has an expression which can be represented with 2 bit-arrays), we can make a recursive algorithm to determine if a group, entity, or system is contained within a group, regardless of whether it is explicitly or not.

A. Recursive Algorithm

Suppose we have some expression, $expr$, that we want to determine if a node in a component's tree has an expression which contains all of $expr$ (i.e. $expr \Rightarrow expr_{node}$).

Base Case:
If the current group node's expression is of the represent-able form, then it is trivial to determine implication and therefore whether $expr$ is contained in the node, using bitwise operations.

Recursion:
If the group node's expression is not explicit, then expr is contained within the group iff expr is contained within the group's parent's expression (recursion) AND the expression implies that the node's sibling representable expression (which it will always have, since the current group doesn't) is NOT true (i.e. $expr \Rightarrow \neg expr_{sibling}$, meaning none of expr is in its sibling).

This recursion will always terminate since the root of a component's group tree is just its component, which is trivially of the represent-able form. This method of navigation is how we decide where to insert groups, what groups to assign systems to, and what groups to insert the components of entities with some given set of components.

A key optimization is that if we start from the root node, we can traverse down the tree based on whether the left, right, or neither node of the currently processed node contains $expr$. This allows us to implicitly remember that the parent of the children we are checking for containment contains $expr$, since we had to have visited it to reach the current node. We thus don't have to deal with checking if the parent of a node contains the expression.

An example of the traversal path for this algorithm can be seen in Figure 6.

V. Entity Implementation & Representation

Entities are represented by a handle (an index to used to query component arrays for their components), a unique ID (used to distinguish existing entities from deleted entities with the same handle), and a single component bit array (which specifies what components the entity does and doesn't have). Whenever a component is added or removed from an entity, the entity's old component bit array is used to inform all relevant (True Symbol set) component arrays that the entity's components should be removed from their current groups, which are determined by navigating the group tree using the entity's component bit array. Then the entity's new component bit array is used to insert the entity's new and existing components into their new groups in much the same way.

Entity creation can be thought of as adding components to an entity with no components, and similarly entity deletion can be thought of as removing all components from an entity.

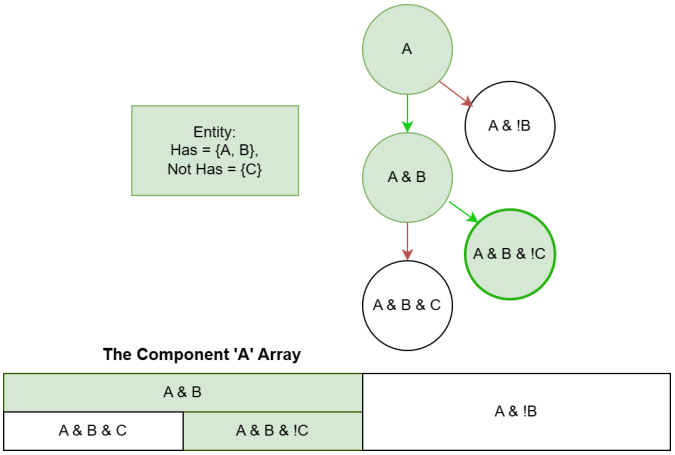

Fig. 6. The traversal path through a group tree for an entity with components A and B. At the root, it is checked if the entity's expression implies the left group expression (A&B); it does so we move to that node. Next, we check if the entity's expression implies the left group expression (A&B&C); it doesn't, so we check if the entity's expression implies the group expression (A&B&C) is NOT true, which it does. Thus, the group for the entity is the group (A&B&!C).

VI. System Implementation & Representation

Systems are represented by a static function pointer, which is called on the components of every entity which matches its component signature, and 2 bit arrays, which together represent its component signature. A system's component signature is the components which the systems cares about and whether the systems requires the cared-about components to exist for an entity or requires them to NOT exist for a entity.

We use C++'s template meta-programming to determine the function pointer arguments and generate a tuple of the component arrays which hold components required by the function pointer. A system makes a request to each component manager for the best fitting group based on its component signature.

Each component array finds the closest-to-leaf group node whose expression is implied by the system's component signature and returns that group to the system. In the ideal case, the group expression and the system's component signature will be equivalent, meaning that the expressions imply each other. When they do, the system can make the very useful assumption that each and every component within

the group will be owned by an entity that should be processed by the system. If only the system's component signature implies the group expression (and not vice versa), the system cannot make that assumption; there could be components in the group which are owned by entities that the system should not operate on.

Full Match:
If all the groups the system receives are equivalent to its component signature and all share the same original BET (i.e. their orderings will be maintained), we consider it a 'full groups system'. Full groups systems can retrieve the starting element for each of its groups (one per component) and iterate over them contiguously without fear of errors or misalignment. This makes full groups systems extremely faster, especially if the compiler is able to inline the system's function pointer call.

Partial Match:
If only some of the groups the system receives are equivalent to its component signature or some groups do not share an original, 'raw' BET, we consider it a 'partial groups system'. Partial groups systems can retrieve the starting element for each of the equivalent groups in the same origin BET, but must utilize entity handle indirection to query component arrays for which it does not have an equivalent, same-BET group.

Blind:
If none of the groups the system receives are equivalent to its component signature, we consider it a 'blind groups system'. Blind groups systems cannot retrieve must iterate through the entities of its smallest size groups and check the entity composition of each entity for whether it satisfies the system's signature, then it must use entity handle indirection to retrieve components for all other groups. This is un-acceptably slow and should be avoided at all costs.

The primary solution for mitigating the performance issues with blind groups systems in other ECS implementations seems to be to cache the components which the blind groups system actually ends up performing operations on into a secondary array; this assumes that entity changes which affect the primary array are infrequent [6] [7].

VII. Code

Code for this project's ECS is maintained in the Github repo, https://github.com/APeculiarCamber/Clique_ECS.
See the ExampleECS.h for a step-by-step example of the ECS's use.

VIII. Evaluation/Experiments

A. Setup

We evaluated our ECS against a more conventional virtual pointer array solution, by creating two programs which compute the position of some number of entities, where each entity has position derivatives up to a specified Nth position derivative (i.e. velocity, acceleration, jerk, snap, crackle, pop, ...). One of the key applications of these derivatives in games is in the simulation of realistic motion and physics. One program uses our ECS system, and the other uses a virtual pointer system.

With some specified maximum Nth derivative, we construct an equal number of entities which only need to compute position from up to the Kth derivative for all k $<=$ N, by randomly selecting a k for each entity. For example, with N=4, we would have a quarter of entities with only Position, a quarter with Velocity and Position, a quarter with Acceleration, Velocity, and Position, and finally a quarter with Jerk, Acceleration, Velocity, and Position.

In the case of the virtual pointer array/, we have an array of virtual class pointers and a pointer for a different implementing class is appended to the array depending on the randomly selected k. In the case of the ECS, we create an entity with the appropriate number of derivatives based on k. Each class is independent of each and computes all new derivatives values, from the (k-1)th derivative all the way to Position.

For ECS, we construct a left-heavy group tree which contains the groups (Position), (Position, Velocity), (Position, Velocity, Acceleration), ..., (Position, Velocity, ..., Nth Derivative). Systems operate on the last 2 derivatives of an entity with at least k derivatives. For example, the system which modifies Jerk based on the current Snap of an entity has a component signature which matches it to the group (Position, Velocity, ..., Snap). This enables all systems to have a group whose expression is fully equivalent to their component signature.

For the virtual pointer array, we attempted to be an generous as possible to it: we ran the virtual pointer array test twice, once maintaining the original appended order (under the assumption that this might improve memory access and thus the pre-fetcher's prediction success rate) and again by ordering the pointers based on there underlying class (under the assumption that the branch predictor will better predict the results of the virtual pointer table lookup for the function call).

To collect our results, we ran the ECS solution and the virtual pointer solution over 1000 ticks of simulating various numbers of entities.

B. Analysis

Figure 7-Left plots the number of entities against the runtime of the ECS solution to take a single tick of the simulation on average, after all setup (grouping and component additions). Predictably, the Derivative N used directly affects the runtime of our ECS solution, since increasing N by one will add another derivative to calculate to the majority of entities.

Figure 7-Right plots the number of entities against the percentage improvement of our ECS solution over the virtual pointer array solution. We note massive speedups for our ECS solution over the virtual pointer solution, especially for lower derivative N's.

It is believed that this is largely due to the time required to make the virtual function lookup and to create stack frames for each virtual function call per entity. Since the ECS solution can have system functions be inlined, neither virtual function lookup tables nor stack frames are required.

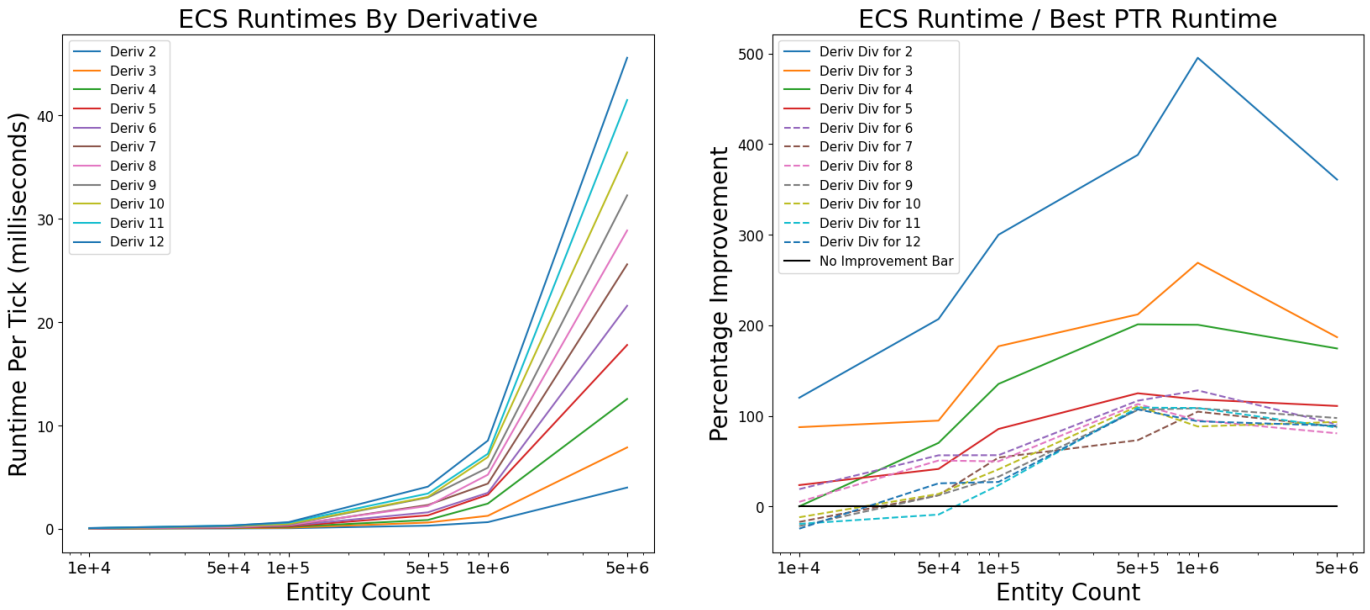

Fig. 7. Left: Runtime for processing speed changes. Right: Performance improvements over the best virtual pointer runtime (in-place or sorted).

At higher derivative counts, the larger amount of computation per virtual function call seems to start to amortize the time required for the virtual lookup and stack frames. Since the computations for all derivatives are performed in the same funciton, the compiler and processor can much better vectorize and take advantage of instruction-level parallelism than the ECS solution (with its separate systems per derative pairs) can. However, the relative performance improvement from amortization also appears to plateau quickly. At high entity counts, 10^6 and $5*10^6$, there is little relative improvement over the ECS solution; the virtual pointer runtimes remain around 80% slower the ECS solution's runtime. This is presumed to be due to the growing cost of memory accesses, since for example, $5*10^6$ entities would constitute dozens of Gigabytes of memory, far more than can be accommodated in any level of cache and so must be saved in Drive.

It is notable that at lower entity counts (10^4), the virtual pointer solution performs better that the ECS solution at higher N's, likely due to the aforementioned amortizations.

Since the organization of components is much more involved than for virtual pointers, the initial creation and adding of components to the ECS is considerably more expensive than the virtual pointer solution.

Fortunately, Figure 8-Left shows that the time required to setup the ECS grows almost linearly with the number of entities. Unfortunately, these times are considerably more than the time required by the virtual pointer system. For example, at N=12, creating $5 * 10^6$ entities for the ECS solution takes 89.7 seconds, while it takes only 0.55 seconds for the virtual pointer solution.

As can be seen in Figure 8-Right, the time lost in setup does eventually amortize when the ECS performs better than the virtual pointer array. In the case of $5*10^6$ entities at N=12 for derivatives, it takes 100.11 seconds (2197 iterations) to amortize after setup.

We have several ideas for improving the performance of the organization stage of our ECS, most interestingly only navigating a single BET on entity changes, rather than navigating one for every component, but we have implemented no optimizations at this stage.

IX. Conclusion

Our group tree paradigm provides what seems to be a novel approach to handling component groups in Entity Component Systems. By leveraging the boolean hierarchical structure of groups and the rightward-only order-of-influence property of maintaining group sub-blocks, we are able to efficiently manage more component groups than typical ECSs.

The evaluation results demonstrate the significant performance of the group tree solution over a virtual pointer solution, especially for high entity counts which require more virtual memory management.

Although we did not have time to benchmark against other ECS implementations, our solution is flexible and adaptable, allowing developers to easily adjust the group hierarchy and query structure to fit their specific needs.

We hope our group tree paradigm offers as valuable a contribution as initial results seem to indicate, and we hope to conduct further research on this, particularly in methods for being less conservative about what groups are allowed and in what order, in caching for partial groups, and in automatic group specification using machine learning. I am very excited about this work.

A. Work After 13th

Not much work on this proejct was done after the 13th: this paper was proofread and polished, and an example C++ file was created to demonstrate the use of the ECS more explicitly than the benchmarks do.

B. Clarifications

I used 'we' and 'our' to refer to myself out of habit and formality. Only Jay Idema worked on this project directly, with massive indirect assistance from Bin Ren.

X. Related Work

Structure of Arrays (SoA) is a data layout that stores multiple arrays of the same data type, with each array containing a single component of a complex data type; this is an alternative to an array of structures approach which is typically considered a more intuitive but potentially less efficient layout. Similar to the ECS paradigm, this layout organizes structure fields into independent arrays and offers similar benefits in memory access efficiency and parallelization. Ikra-CPP [5] is a C++/CUDA DSL for Object-Oriented Programming which attempts to support a structure of arrays layout for data but providing a more traditional object oriented array of structures method of reasoning about the language. SoAx [4] is similarly a library for defining structure of array layouts and operations with the handiness of array of structure thinking.

Most ECS research appears to be rather recent, most within the last 5 years. Vico [1] is a co-simulation framework for an Entity-Component-System which describes a concept of 'families' similar to my 'groups' in this paper. Investigations into the similarities and differences between ECS use in the domains of interactive military-centric simulators and interactive computer games have been conducted [3], indicating promising opportunities for possible applications outside of interactive computer games. There are numerous popular open-source implementations; EnTT [6] and Flecs [7] are among the most popular C++ ECS on GitHub and their API documentations were invaluable in understanding the degree and design of abstractions for these systems.

Halide [8] is a domain-specific language for image processing that allows developers to write high-performance image and array processing pipelines that can be compiled for a range of different hardware targets. The key innovation of Halide is the separation of algorithm from scheduling. Programmers write declarative code that specifies what should be computed, which is seperate from the scheduling code that specifies how computations should be scheduled onto hardware targets.

The latest Halide autoscheduling paper [9] describes an autoscheduling algorithm for Halide that uses machine learning to generate optimized schedules for Halide programs. The machine learning algorithm generates schedules by searching through a space of possible schedules using a combination of tree search and random program generation. This makes Halide more accessible to programmers without optimization expertise and makes it easier to port Halide programs to different hardware targets. We are very interested in applying a similar 'auto-grouping' concept to future work.

References

[1] L. I. Hatledal, Y. Chu, A. Styve, and H. Zhang, "Vico: An entity-component-system based co-simulation framework," *Simulation Modelling Practice and Theory*, vol. 108, p. 102243, 2021.

[2] D. Wiebusch and M. E. Latoschik, "Decoupling the entity-component-system pattern using semantic traits for reusable realtime interactive systems," in *2015 IEEE 8th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*. IEEE, 2015, pp. 25–32.

[3] D. D. Hodson and J. Millar, "Application of ecs game patterns in military simulators," in *Proceedings of the International Conference on Scientific Computing (CSC)*. The Steering Committee of The World Congress in Computer Science, Computer ..., 2018, pp. 14–17.

[4] H. Homann and F. Laenen, "Soax: A generic c++ structure of arrays for handling particles in hpc codes," *Computer Physics Communications*, vol. 224, pp. 325–332, 2018.

[5] M. Springer and H. Masuhara, "Ikra-cpp: A c++/cuda dsl for object-oriented programming with structure-of-arrays layout," in *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*, 2018, pp. 1–9.

[6] M. Caini, "Entt: Ecs implementation," *https://github.com/skypjack/entt*, 2017.

[7] S. Mertens, "Flecs: Ecs implementation," *https://github.com/SanderMertens/flecs*, 2018.

[8] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[9] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand *et al.*, "Learning to optimize halide with tree search and random programs," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 4, pp. 1–12, 2019.

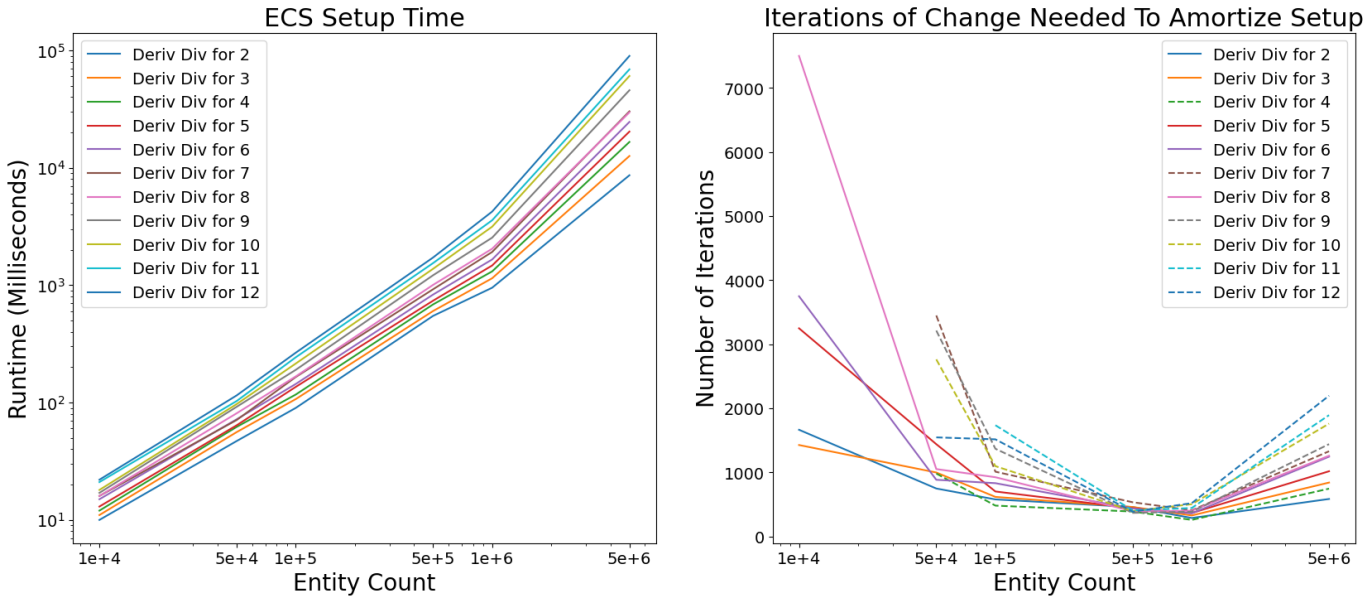

Fig. 8. Left: Time (milliseconds) taken to setup up the ECS for various entity counts. Both the x and y axis are in log scale, demonstrating the almost linearly relationship between entity count and setup time. Right: The number of iterations needed to amoritize graphed against the number of entities for different last Nth derivatives. The ECS amortizes in the fewest iterations for $5 * 10^5$ and 10^6 entities.