BURSTED BEHAVIOR TREES

Jiří Berný
Jakub Gemrot
Department of Software and Computer Science Education
Univerzita Karlova
Matematicko-fyzikální fakulta
Ke Karlovu 3, 121 16 Praha 2
Czech Republic
email: berryjee@gmail.com
email: gemrot@gamedev.cuni.cz

KEYWORDS
Behavior trees, Implementation, Performance, Unity, Job system, Burst, Visual editor

ABSTRACT

We present Bursted Behavior Trees (BBT) for Unity, which makes use of Unity's Job system, Burst compiler, and is complemented with a node-based visual editor. BBT is currently the only behavior tree solution for Unity capable of execution on Unity worker threads while being Burst compiler compatible and featuring a node-based visual editor. A user can easily create not only custom leaf nodes (sensors and actions) but also inner nodes (composites and decorators) implementing custom behavior selection strategies. We discuss technical challenges implementing such a solution and compare its core features and performance with two popular behavior tree Unity assets. We show that BBT allows for the execution of more trees per frame while being more extendable than its selected competitors.

INTRODUCTION

Games are prevalent not only to human species (Bekoff and Byers 1998) and there is no wonder that they found their way to computers as soon as they could (Russell et al. 1962) followed by other digital devices consequently (Hurley and Wald 2020). As computers grew stronger, more complex games were produced some providing players with large open-worlds to roam in. To make these worlds alive, creators populates them with human-like characters, referred to in the literature as non-player characters (NPCs), intelligent virtual agents (IVAs) (De et al. 2001) or animats (Wilson 1991). There are many ways how to encode behaviors of these characters, for which symbolic techniques are prevalent many of which are falling under the umbrella of dynamic or reactive planning. Reactive planning has roots in robotics in the eighties (Georgeff and Lansky 1987) where it was pit against classical planning. Classical planning was under-performing at that time as computational resources of computers were modest compared to contemporary machines. One of the advantages of reactive planning wrt. classical was its fast execution. Instead of computing a whole plan how to achieve sought goal, a reactive plan looks for "the next action to execute", which is done in "timely fashion" (Bryson and Stein 2000). This property appeals also to game developers as games are soft real-time systems (Buttazzo et al. 2005) and NPC behaviors must execute fast even today (AI has about 10perc. CPU budget in games; for illustration, simulating 10 NPCs 10x per second means each NPC has at max 1ms to output its action, which may even require path-finding and spatial awareness). Out of the various approaches, e.g., scripting, ad-hoc goal-oriented architectures, different variations of finite state machines, etc., behavior trees (BT) (Isla 2005) are frequently used today. They are part of various mainstream game engines such as Unity, Unreal Engine 4 (UE4), Godot, and others; they are either built-in (UE4) or available as plugins (Godot) or downloadable assets (Unity).

Contribution

In this paper, we present Bursted Behavior Trees (BBTs). BBT is a behavior tree solution tailored specifically for the Unity engine. It features time-efficient implementation utilizing Unity's worker threads and Burst compiler (Unity Technologies 2021a) while still being easy to work with. It allows for simple behavior tree creations via the visual editor and allows for creating custom behavior tree nodes via clutter-free C# node implementation. The (time) performance is compared to the most popular paid and free behavior tree solution assets obtainable through Unity Asset Store, Behavior Designer (Opsive 2014), and Panda Behaviour Free (Begue 2016).

Our contribution is three-fold. First, our solution is the only solution that executes on the Unity job threads, thus allowing to run N-times more behavior trees than

popular solutions as it scales with the number of processor cores available. Second, our solution supports the simple creation of custom behavior tree nodes allowing developers to customize the BT solution easily; a feature popular BT assets are lacking. Third, our technical solution is uncommon among BT implementations as we are using code generators for transpiling behavior tree definition into C# code. It allows us both to mask the complexity of running behavior trees on Unity job threads and to speed up their execution (both behavior tree initialization and execution) at the same time.

Structure

The rest of the paper is organized as follows. First, we review the definition of behavior trees (BT) and present related work listing requirements on modern BT solutions. Second, we discuss the Unity technical background required for the description and evaluation of our solution. Third, given the technical consideration of the previous section, we describe challenges we had to overcome during the implementation of our solution. Fourth, we compare our solution with selected competitors and present and discuss the results of performance tests. Finally, we conclude the paper by summarizing the key strengths of BBT.

RELATED WORKS

Behavior trees emerged from the video game industry during the talk on artificial intelligence in Halo 2 in 2005 (Isla 2005). They were then popularized in 2007 (Champandard 2007) and went the mainstream in the industry since. In 2012, the Behavior Tree Starter Kit for C++ was published (Champandard and Dunstan 2012), whose implementation was revisited in 2019 (Champandard and Dunstan 2019).
Behavior trees (BTs) as a term refer more to a general idea than a standardized computational model Here we understand BT as a directed tree consisting of inner nodes and leaves. Leaves are either boolean sensors used for context checks or durative actions to be executed by an agent within the environment. Inner nodes arbitrate the execution of their children, which are either decorators (single child only) or composites (one or more children). During execution (tick), BT is traversed from the root. Each node either selects a child for execution or returns its execution result (success, failure, or running). This continues until the execution result of the root is determined. For efficiency, behavior tree execution can skip evaluation of certain nodes if their state has not changed since the last tick; such behavior trees are called event-driven behavior trees (Champandard and Dunstan 2019).
BTs are used predominantly by the video game industry for modeling NPC behaviors (Sekhavat 2017), but they found their way to other fields as well. BTs are used in many robotic projects today (Ghzouli et al. 2020) as, e.g., a replacement for state charts and activity diagrams, or as tools for modeling behaviors of soldiers (Francillette et al. 2020), bicycle riders (Ding et al. 2020) or seniors for the purpose of smart home simulations (Bouchard et al. 2018).
Another advantage of behavior trees is their openness and adaptability. There are quite a few behavior tree extension proposals including new types of nodes improving cooperation between agents (Agis et al. 2020), realizing contextual late-binding of sub-trees (Flórez-Puga et al. 2009) or prioritizing behaviors according to an emotional state of an NPC (Johansson and Dell'Acqua 2012) and others.
However, papers detailing concrete implementations, apart from (Champandard and Dunstan 2019), are rather scarce. Zielinski provides insights into the implementation of bots for the Paragon game (Zieliński 2019) mentioning new node types and extreme parameterization. In the ORION framework, (Buche et al. 2018) authors are detailing the implementation of the system for creating BTs for NPCs based on imitation of human players (Buche et al. 2018). In AIPaint (Becroft et al. 2011), authors describe an authoring tool for BTs designed for non-programmers, though from the high-level perspective only.

Solution requirements

According to the landscape of both existing BT solutions in modern game engines and research articles, we identify two categories of requirements on a BT solution: usability and performance. Usability features are extensibility (allows to create custom BT nodes, i.e., sensors, actions, and inner nodes), BT nodes parameterization, and visual editor for their design. Performance features required are fast instance initialization, fast and zero-garbage execution, scalability with CPU cores.

TECHNICAL BACKGROUND

The main performance limitation of solutions developed for Unity stems from its design. Initially, Unity was implemented as a thread-per-subsystem engine (Gregory 2018). Rendering, physics, audio have their threads but all the game code is executed on the single (so-called main) thread. It is possible to spawn custom threads but most Unity systems are not thread-safe. This requires a user to use Unity APIs from the main thread only, which is shared with critical Unity engine code. Time-demanding game code can limit the game frame rate. Unity Software Inc. (developer of Unity) is well aware of this problem. As a response, they introduced Data-Oriented Technology Stack (DOTS) in 2018, which is being incrementally developed since.
Data-Oriented Technology Stack (DOTS) is a set of tools for writing high-performance, multi-threaded, race

condition-free, data-oriented code in Unity. (Unity Technologies 2021b) This allows for utilization of additional CPU cores. However, developing games with DOTS is more technical than writing single-(main-)threaded code, and existing (both native and 3rd party) features and assets are ported to DOTS slowly.
We now summarize technical challenges connected with developing for DOTS focusing on limitations of developing for Unity job system and Burst compiler we utilize in BBT.

Unity Job system and jobs

Unity Job system manages (so-called) worker threads, which can be used through its API only. A new job is created by declaring a new structure implementing one of the provided job interfaces. To schedule a job for execution, a new instance has to be created, its member fields initialized, and its Schedule method called providing job dependencies. The Job system then creates a copy of the job and adds it to a job scheduler for execution when appropriate. Since the execution is done on a copy, all persistent data has to be stored on a native heap via Unity-provided native collections. A job can contain either managed code or unmanaged code. The latter has a key advantage that the job can be compiled by Burst compiler into a native code for targeted hardware for achieving higher performance (Unity Technologies 2021c).

Burst compiler

Burst compiler is Unity's custom compiler that translates IL/.NET bytecode to native code using LLVM performing various optimization in the process (e.g. method inlining, SSE instructions, and others). However, code produced by the Burst compiler uses only the memory stored in the native (non-garbage collected) heap or stack and works only over the subset of the C# language. Implied limitations wrt. this paper are: 1. Burst does not support any methods related to managed objects, which prevents the use of non-static classes including managed arrays and collections (all data and the code are organized around structures); 2. C# does not support inheritance of structures; 3. interfaces can be used as long as they do not require boxing (C# 8.0 comes with Interfaces with default implementation, but it should not be used with structures, because it causes hidden boxing). 4. delegates can be used with Burst indirectly only via the use of a custom FunctionPointer structure.
All these points prevent the implementation of many design patterns used in object-oriented programming (Johnson and Vlissides 1995) as these rely on inheritance and a virtual function calls. For instance, a BT node class from the Behavior Tree Starter Kit (Champandard and Dunstan 2019) is declaring virtual method that implements the node's execution. This method is then overridden in subclasses to provide custom behavior selection strategies. An approach we cannot use because of Burst.

Native collections and allocations

Unity provides so-called Native variants to many regular C# collections. The main difference is that Native collections are allocated on an unmanaged heap and can hold only unmanaged data. Unfortunately, Unity Native collections are not unmanaged, as the result, e.g. a Native array of Native arrays cannot be declared. Moreover, Native collections can be allocated only on the main thread, before the job execution, and there can be no other dynamic allocation during job execution.

Fortunately, since the collections operate with unmanaged memory, we can safely store pointers to them. A feature our solution relies heavily on to make up for limitations imposed by Burst.

Function pointers

To make up for the missing support of delegates in Burst, Unity provides a FunctionPointer structure (FP) as a realization of function pointers. They can be used to make up for missing virtual calls in Burst-compiled code though there are a few limitations. 1) They can hold a pointer onto static methods only, 2) an FP must always be compiled from a managed code, 3) a function wrapped with an FP cannot return a value, 4) Burst version we use (1.5.3) does not support FPs parametrized by a generic delegate, but version 1.6.0 preview 2 brings support for this type of delegates. As a result data and functions operating over them must be declared separately in the spirit of the old C approach.

Instability

Burst compiler is still in development. We have encountered problems with compiling FunctionPointers from generic static methods or compiling FPs from static methods of generic structs inside a generic method.

IMPLEMENTATION

We now detail several design decisions we have to make in order to reach our solution requirements. 1) How to represent node types with limitation to structures without inheritance with the respect to extensibility. 2) How to allocate and initialize tree data that have to be stored in the native heap. 3) How to allow for parameterization of nodes. 4) How to allow for user-writable generic blackboard for storing intermediate values during BT execution.

Memory allocation

Problem. Tree nodes may contain their inner state that has to be persisted between ticks, different node types require storage of different states. Jobs are executed on a copy of a job structure, we need to limit the amount of data that is necessary to copy.
Solution. We rely on the fact the behavior tree is constructed during design time and thus we know the exact structure of the tree. As the result, each BT node may preallocate a space required to store its internal state. All such states can be then kept in a single NativeArray<byte> we call UniversalNativeArray (UNA). UNA allocates the memory in one call and, since the memory is allocated on an unmanaged heap, the position of the memory is preserved until deallocation. Each node can therefore receive a pointer to a memory where its data is stored.

Node representation

Problem. Since behavior tree nodes may be of different types with different logic, tree execution code cannot realize nodes' logic directly. Inheritance (and thus virtual functions) cannot be used with structures in C# and we cannot use interfaces as it leads to boxing.
Solution. We designed a NodeHandle, which binds together data with a function that operates upon them. NodeHandles can be assembled according to the needs of the respective trees during runtime. As we already have nodes data stored in UNA we can declare static methods that operate over those data and therefore we can use FunctionPointer to make up for missing virtual calls. NodeHandle then combines a pointer onto node's data and node type specific FunctionPointer that is to be invoked upon them BT tick.

Code generators

Problem. BT solution needs to be extensible; a user must be able to define custom sensors, effectors, and inner nodes with custom behavior selection logic. However, a custom node implementation requires code duplication as inheritance cannot be used.
Solution. We implemented a code generator that, given the node description in the form of C# class, generates both structures required for node execution as well as classes required for visual editor realization.
We use the Text Template Transformation Toolkit (shortly T4). T4 is a toolkit for text generation, that is using text templates. Text templates are composed of text blocks and control logic.(Partlow et al. 2016) We created a few code generators: a NodeGenerator, an EditorGenerator, and a TreeGenerator. Fig. 1 shows how the user input in the form of a NodeDeclaration, a C# class (partial example in the Listing 1.), is transformed down to a TreeJob.

NodeGenerator is used to generate node structures (`BBT_Nodes`), which declare node execution methods. Everything the user has to do is to write an implementation of the behavior selection logic along with the required node state.

EditorGenerator takes the same input as NodeGenerator and produces a representation of the node for the visual editor.

Given the tree assembled visually via the TreeEditor, TreeGenerator generates a code that is initializing BT as a struct using `BBT_Nodes` of nodes used within the tree. Apart from node execution, this code also allocates the memory for the tree, initializes nodes, sets up a tree blackboard (BB), and initializes the input parameters of nodes as per design by the user.

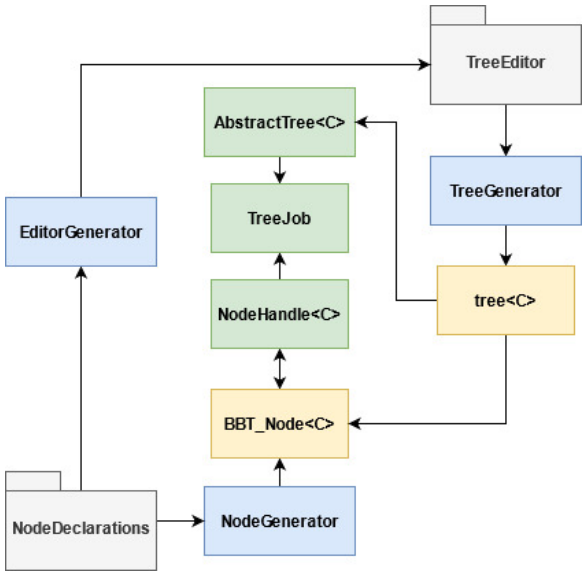

Figure 1: The architecture of trees and nodes.

Context and blackboard

Problem. BT sensors are frequently accessing game-specific values to control the execution of behavior trees. Moreover, it is advantageous to be able to declare, read and write variables that are private for every tree instance executed (similarly to private object fields). However, only unmanaged data apriori bound to a Unity job can be queried from within the job.

Solution. A user specifies a BT context upfront and is given a hook within the code to pass data from the main thread to UNA of the tree, which can be accessed by BT nodes from within the job. A similar approach is used for blackboard that can allocate enough memory in UNA to store its content during tree instance initialization.

Table 1: Usability features overview.

	BBT	PB	BD
Custom node	any	leaves	leaves
Visual editor	yes	no	yes
Node params	yes	no	yes
Input params	any	constant	constant, BB value

Input parameters and Sensors

Problem. BT nodes may require input parameters, which are frequently specified via the visual editor and can have a form of C# function calls. However, the exact locations of these functions must be known during compile time.
Solution. A user is allowed to define custom static methods, which the visual editor will automatically provide code completion for. Our solution then automatically compiles FunctionPointers to all such functions used. The inability of FunctionPointers to return values is circumvented by passing a pointer to return value reserved within UNA, which the sensory function is ought to fill instead of returning the value. Additionally, we allow these sensory methods to interact with BT context and BT blackboard.

COMPARISON - USABILITY

In this chapter, we discuss the usability features of our solution, Panda Behaviour (PD) and Behavior Designer (BD) wrt. to categories identified in the RELATED WORKS section.

Custom BT Nodes

Panda Behavior supports the definition of custom sensors and actions only called "tasks" by PD. Users are not allowed to create custom composites and decorators.
Behavior Designer supports the definition of custom sensors and actions called Conditional tasks resp. Action tasks by PD. Users are not allowed to create custom composites and decorators.
Our solution supports the definition of custom nodes of any category (composites, decorators, sensors, and actions).
To create a new node, a user defines a new structure type and uses provided C# attributes to describe node category and input parameters. Composites can optionally specify the fixed amount of children nodes. NodeGenerator then generates a partial structure with the same name, which has solved complicated initialization, data mapping, and properties for easier access of NodeHandle's data, and resolving input parameters. A user provides only the body of event methods without knowledge of internal functionality. An example of such a custom node is given in the List. 1 that is describing "if-then-else" node (for illustratory purposes).

```
[BBT_Node("Composite")]
[Active]
public struct IfElse
{
    [InputParameter]
    public bool Predicate;

    public Child IfChild;
    public Child ElseChild;
}
```

Listing 1: Custom node example

Node parameterization

Panda Behaviour allows only for constant values as arguments to nodes, i.e., not supporting parameterization at all.
Behavior Designer allows for constant values and blackboard item references to be passed as arguments to nodes. The use of blackboard items allows for parameterization. However, it does not allow the use of custom functions for computing the arguments on the fly.
Our solution allows parameterizing nodes using even custom C# function calls. Moreover, custom functions may require their own parameters, which can be recursively specified.

Visual editor

Panda Behaviour trees are defined as the text files with structure defined by the text indentation. These files are parsed in the run-time during the tree initialization.
Behavior Designer has an integrated visual editor and the trees are stored in asset files and parsed in run-time.
Our solution contains the integrated visual editor, which generates C# code implementing the tree directly. As the result, we do not have to read files or use reflection to initialize a tree in run-time. This approach is much faster because we only allocate the data and connect the children with parents. FunctionPointers required by the tree are compiled only once per node type.

COMPARISON - PERFORMANCE

To compare the performance of our solution with selected Unity assets, we designed tests to measure BT time and memory efficiency. We measured: build times, dynamic memory footprints, execution time (for both small and large trees).
We constructed a few trees of different sizes for all three compared solutions respectively. Trees were built out of the following nodes: Sequence, Repeat, Succeed, and SucceedNext. A Sequence node is a composite node that executes its children in sequence. Repeat node is a decorator node that keeps executing its child regardless of

its result. Succeed node is an action node that always returns the success result upon execution. SucceedNext node returns the running result during the first execution and the success result on consecutive executions.

Large and Small (Succeed) Trees

For execution time and memory footprint tests, we have created full quaternary trees with Sequence nodes as inner nodes and Succeed nodes as leaves . To measure multiple ticks we have placed this tree as a child of the Repeat node. The performance was measured on two trees of two sizes: 342 nodes for a small tree and 5462 nodes for a large tree. Those trees are having the property that they are fully traversed in a depth-first manner during BT tick simulating the high workload of the tree.

Binary (SucceedNext) Tree

For performance stability tests, we have constructed similar trees but of different proportions. Each Sequence node had only two children and we used SucceedNext nodes as leaves. The size of the tree was 2048 nodes. These trees mimic more realistic BT executions as their evaluation finishes as soon as the running result of the hit SuceedNext node is propagated back to the root. The SucceedNext node is meant to simulate the execution of a short durative action as it returns a running result during its first tick and succeeds in the second.

Testing Environment

We have performed all tests on a laptop with the Intel Core i7-9750H (6 cores, 12 threads) processor and 16 GB RAM. Unity has reserved 11 worker threads. Unity version 2021.1.0f1, Burst version 1.5.1, Entity Component System version 0.17.0, Panda Behaviour version 1.4.3, Behavior Designer version 1.6.7, Windows 10 - version 20H2, OS build 19042.870 were used. Burst compiler was enabled to perform synchronous compilation.

Build Time Test

Aim. In this test, we measure the time that is required to initialize trees. High initialization times may create CPU spikes negatively affecting certain simulation frames causing, e.g., a drop in the framerate and thus tearing in visualization. Our solution does not use any reflection or tree parsing during tree initialization, therefore we expect significant performance improvement in contrast with PB and BD.

Procedure. We use large trees with 5462 nodes in batches of several sizes from 10 up to 5000 instances. The Y-axis represents the total running time.

Results. The Panda Behaviour builds trees in the first tick. Behavior Designer has two options. It can either build trees in their first tick or build them asynchronously in a separate thread. However, we can't check if a tree is still loading. Because of this reason, we measure the building time of the trees, together with the first tick and we subtract the average tick time. Since our solution builds the tree in the code, it is substantially less power consuming as can be seen in Fig. 2.

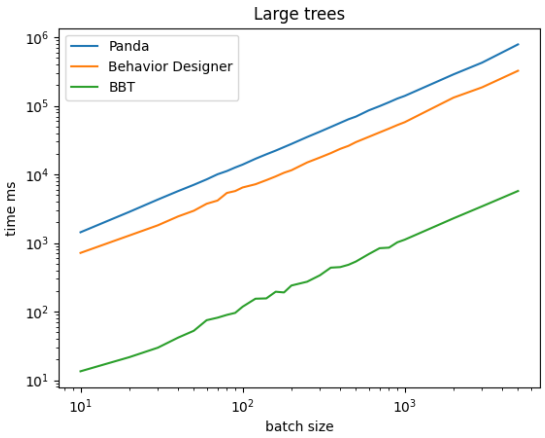

Figure 2: Build time with large trees.

Memory Footprint Test

Aim. This test aimed to compare the amount of memory that is allocated for trees.

Procedure. We use large trees with 5462 nodes in batches of several sizes from 10 up to 5000 instances. The Y-axis represents the amount of used dynamic memory in MB. The measurement was taken after the tree building and after the forced garbage collection. For our solution, we have summed up the amount of managed and unmanaged memory.

Results. Looking at Fig. 3, our solution requires much less memory.

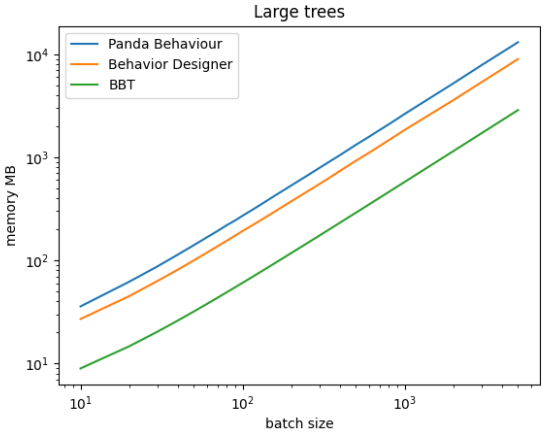

Figure 3: Allocated memory measured for trees.

Execution Time Test

Aim. This test aimed to compare the time that is required to execute trees.

Procedure. We use large trees with 5462 nodes and small trees with 342 in batches of several sizes from 10 up to 5000 instances. The Y-axis represents the total time that is required to tick all trees from the batch. In our solution, we measure separately the time required to schedule jobs on the main thread and the time that is required to process all trees in parallel on worker threads. For scheduling, we tested two strategies. The batched schedule prepared all jobs and schedules them on worker threads in one call while the ordinal schedule prepares a job and schedule it on a worker thread right after which is repeated for all trees in a given batch.

Results. Looking at Fig. 4, the Behavior Designer has serious performance problems even with lower tens of trees. Users can optionally limit the number of ticked nodes per frame, nevertheless, it can be hard to balance when different trees are ticked per frame. The Behavior Designer has event-driven execution, which can bring unwanted overhead. That is because in this case during each tick we have to go through all of the nodes. The Panda Behaviour is approximately 5x faster. Our solution is even 5x faster than Panda Behaviour. If we look at the scheduled time, we can see, that scheduling an even higher number of trees takes a negligible amount of computation power, in comparison to the single-threaded solutions.

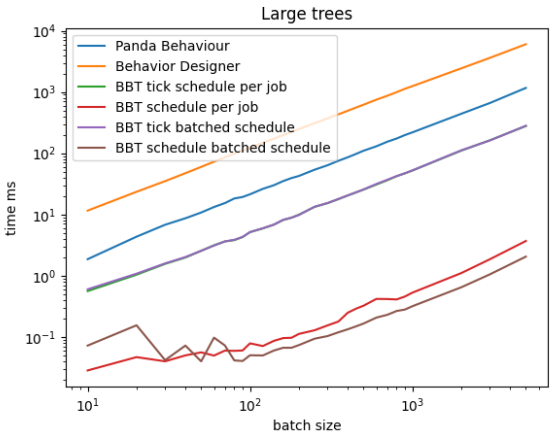

Figure 4: Tick performance on big trees.

When we look at the test with small trees (Fig. 5), we can see, that the scheduling time is getting closer to the actual execution. The difference between the Panda Behaviour and our solution is smaller for the low number of trees, but we can still get a much lower workload for the main thread. If we look at scheduling strategies, we see that batched scheduling is faster if the workload is higher but more unstable with a lower workload.

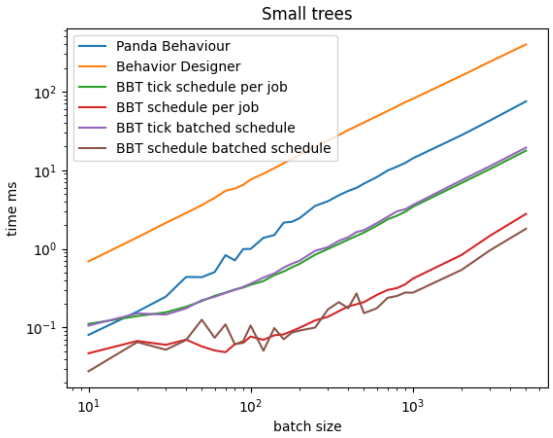

Figure 5: Tick performance on small trees.

CONCLUSION

In this paper, we presented Bursted Behavior Trees (BBTs) for Unity. It's the only BT solution that makes use of the Unity Job system and Burst compiler while being extendable, featuring a visual editor, and allowing for node parameterization. We compared BBT with two popular BT Unity assets: Panda Behaviour Free and Behavior Designer. We showed our solution has lower BT initialization times, scales with CPU cores, is overall faster in certain scenarios while being more extendable than competitors.

FUTURE WORKS

Scheduling brings significant overhead; it might be useful to tick more trees in one job. Blackboard and node input parameters support custom unmanaged data types, but the user can not use native collections, because the TreeGenerator does not allocate them.

ACKNOWLEDGEMENT

This work was supported by the Charles University grant SVV-260588.

REFERENCES

Agis R.A.; Gottifredi S.; and García A.J., 2020. *An event-driven behavior trees extension to facilitate non-player multi-agent coordination in video games. Expert Systems with Applications*, 155, 113457.

Becroft D.; Bassett J.; Mejía A.; Rich C.; and Sidner C., 2011. *Aipaint: A sketch-based behavior tree authoring tool.* In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. vol. 6.

Begue E., 2016. *Panda Behaviour*. URL https://assetstore.unity.com/packages/tools/ai/panda-bt-free-33057.

Bekoff M. and Byers J.A., 1998. *Animal play: Evolutionary, comparative and ecological perspectives*. Cambridge University Press.

Bouchard B.; Gaboury S.; Bouchard K.; and Francillette Y., 2018. *Modeling human activities using behaviour trees in smart homes*. In *Proceedings of the 11th PErvasive Technologies Related to Assistive Environments Conference*. 67–74.

Bryson J. and Stein L.A., 2000. *Architectures and idioms: Making progress in agent design*. In *International Workshop on Agent Theories, Architectures, and Languages*. Springer, 73–88.

Buche C.; Even C.; and Soler J., 2018. *Autonomous virtual player in a video game imitating human players: the ORION framework*. In *2018 International Conference on Cyberworlds (CW)*. IEEE, 108–113.

Buttazzo G.; Lipari G.; Abeni L.; and Caccamo M., 2005. *Soft Real-Time Systems*. Springer.

Champandard A.J., 2007. *Behavior trees for Next-Gen AI. Game Developers Conference Europe*.

Champandard A.J. and Dunstan P., 2012. *The Behavior Tree Starter Kit*. URL https://github.com/aigamedev/btsk.

Champandard A.J. and Dunstan P., 2019. *The behavior tree starter kit*. In *Game AI Pro 360*, CRC Press. 27–46.

De A.A.; Aylett R.; and Ballin D., 2001. *Intelligent virtual agents: third international workshop, IVA 2001, Madrid, Spain, September 10-11, 2001*. Springer.

Ding W.l.; Ding X.; Chen K.; Wan Z.x.; Xu Y.; and Feng Y.j., 2020. *A Computer Model for Simulating the Bicycle Rider's Behavior in a Virtual Riding System. KSII Transactions on Internet & Information Systems*, 14, no. 3.

Flórez-Puga G.; Gomez-Martin M.A.; Gomez-Martin P.P.; Díaz-Agudo B.; and González-Calero P.A., 2009. *Query-enabled behavior trees. IEEE Transactions on Computational Intelligence and AI in Games*, 1, no. 4, 298–308.

Francillette Y.; Bouchard B.; Bouchard K.; and Gaboury S., 2020. *Modeling, learning, and simulating human activities of daily living with behavior trees. Knowledge and Information Systems*, 62, 3881–3910.

Georgeff M.P. and Lansky A.L., 1987. *Reactive reasoning and planning*. In *AAAI*. vol. 87, 677–682.

Ghzouli R.; Berger T.; Johnsen E.B.; Dragule S.; and Wasowski A., 2020. *Behavior trees in action: a study of robotics applications*. In *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. 196–209.

Gregory J., 2018. *Game engine architecture*. crc Press.

Hurley L. and Wald H., 2020. *13 things that prove that DOOM will run on literally anything*. URL https://www.gamesradar.com/12-things-that-prove-that-doom-will-run-on-literally-anything.

Isla D., 2005. *Gdc 2005 proceeding: Handling complexity in the halo 2 ai. Retrieved October*, 21, 2009.

Johansson A. and Dell'Acqua P., 2012. *Emotional behavior trees*. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 355–362.

Johnson R. and Vlissides J., 1995. *Design patterns. Elements of Reusable Object-Oriented Software Addison-Wesley, Reading*.

Opsive, 2014. *Behavior Designer*. URL https://assetstore.unity.com/packages/tools/visual-scripting/behavior-designer-behavior-trees-for-everyone-15277.

Partlow J.; Mabee D.; Jacobs M.; Turn N.; Warren G.; maqiv; Hogenson G.; Jones M.; Homer A.; Cai S.; and et al., 2016. *Code Generation and T4 Text Templates - Visual Studio*. URL https://docs.microsoft.com/en-us/visualstudio/modeling/code-generation-and-t4-text-templates.

Russell S.; Graetz M.; and Witaenem W., 1962. *Spacewar. Computer software*.

Sekhavat Y.A., 2017. *Behavior trees for computer games. International Journal on Artificial Intelligence Tools*, 26, no. 02, 1730001.

Unity Technologies, 2021a. *Burst User Guide: 1.6*. URL https://docs.unity3d.com/Packages/com.unity.burst@1.6/manual/index.html.

Unity Technologies, 2021b. *DOTS packages*. URL https://unity.com/dots/packages.

Unity Technologies, 2021c. *What is a job system?* URL https://docs.unity3d.com/2021.2/Documentation/Manual/JobSystemJobSystems.html.

Wilson S.W., 1991. *The animat path to AI*.

Zieliński M., 2019. *Paragon Bots: A Bag of Tricks*. In *Game AI Pro 360*, CRC Press. 83–94.