HITMAN* 2: Enhancing Reverb on Modern CPUs

Introduction

Popularization of CPUs with six and eight cores means that untapped processing power becomes available to games, and part of that power can be used to enhance our players' audio experience. To take advantage of extra cores, developers need to aggressively parallelize code execution and make sure game features scale correctly on different CPUs. This paper, based on a joint effort by IO Interactive and Intel, demonstrates how we used multicore CPUs and parallelization to make Hitman* 2 sound more immersive. More specifically, we talk about improving an important and computationally expensive area of game audio—the reverb.

About HITMAN* 2

Hitman 2, a stealth video game developed by Copenhagen-based IO Interactive and released in November 2018, is a continuation of the franchise that has attracted millions of fans around the world. Our players, assuming control of the iconic assassin Agent 47, explore large and vibrant levels, and find creative ways to flawlessly eliminate important and well-protected targets.

Embedded official HITMAN2 release trailer

Whether to discreetly gain access to the target while disguised as another game character, orchestrate an accident, or complete the mission with guns blazing is entirely up to the player. The game provides ample opportunities for any approach, as well as the ability to play with or against another person in Coop and Versus multiplayer modes. Hitman 2 is a live universe that keeps growing as new locations, missions, and content are added with regular updates.

The game is developed using the proprietary Glacier game engine and Audiokinetic Wwise* as the audio middleware. Many implementation details in this article will thus be specific to Wwise, but they can easily be adapted to any sound engine.

Simple Reverb System and Its Drawbacks

Reverb is the phenomenon that is created when the soundwaves are reflected off the surrounding surfaces. These reflections amplify and *color* sound, and cause it to persist after the original, direct sound waves have faded. In video games, reverb is an important immersion tool that gives players aural clues about environments where the action unfolds. Given the myriad of reflections that contribute to a reverberated sound, precise simulation of this phenomenon is very computationally expensive; therefore, game developers must resort to various approximations.

A popular approach is to spatially subdivide a game world into separate areas and assign a reverb preset to each area. This article refers to such areas as *rooms*, for consistency with Glacier and Wwise terminology, even though they do not necessarily represent enclosed spaces. A room in this context is a part of the game level with similar acoustic properties. Consequently, rooms roughly imitate physical structure of the level, representing both indoor and outdoor environments, and even parts of open terrain. In Hitman 2 they are defined using 3D shapes such as bounding boxes, cones, and convex polygonal shapes, as shown in Figure 1.

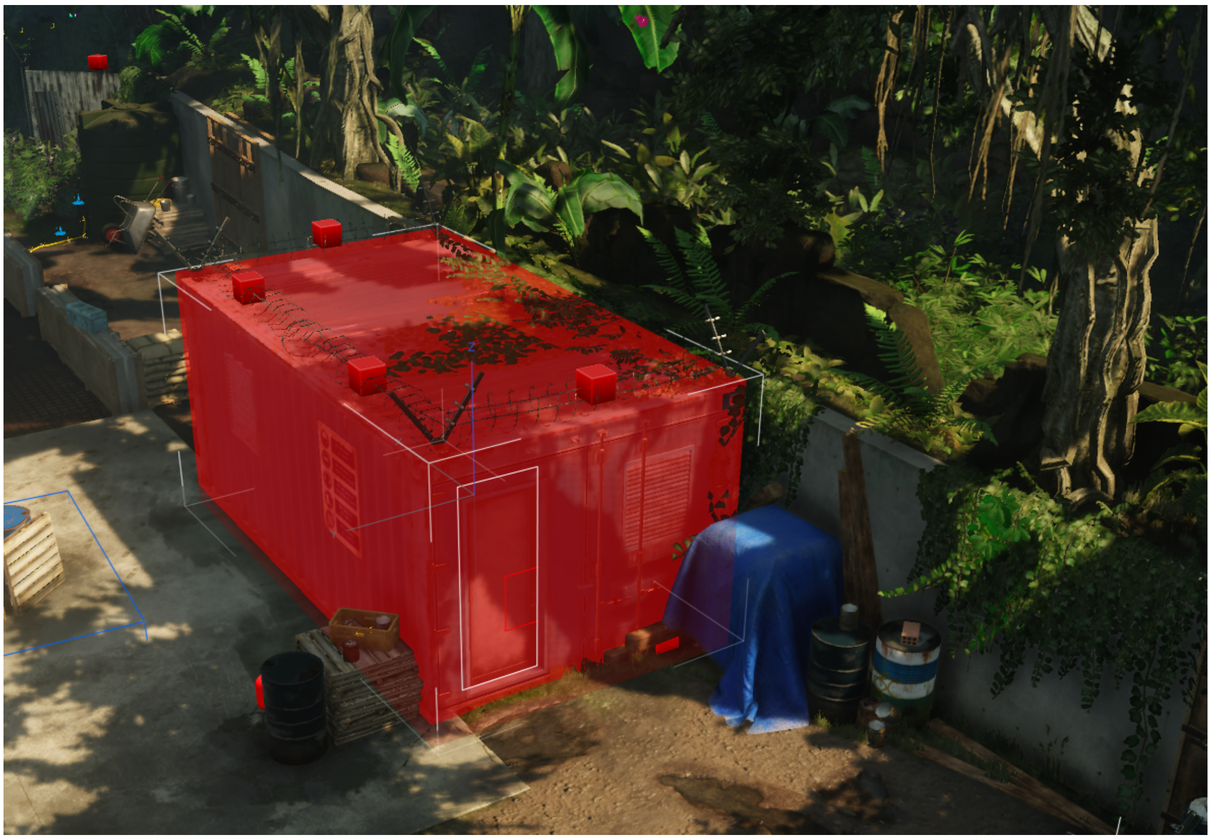


Figure 1: A very simple example of sound geometry. In this case, the entire hut is represented by one sound room highlighted in red.

A reverb preset is a reverb effect with predefined settings that are fine-tuned to simulate acoustics of a specific room. The effect used may be synthetic reverb (such as Wwise RoomVerb) or convolution reverb. Hitman 2 mostly uses the latter.

Synthetic effects simulate reverberation algorithmically by using delay lines and filters such as *comb* and *all-pass*. These effects use less CPU, but they tend to sound less natural and aesthetically pleasing than convolution reverb, especially for rooms with complex acoustics and longer reverberation time. They also tend to be harder to set up to achieve satisfying results.

Convolution reverb, on the other hand, uses a prerecorded impulse response of a real-life location. An impulse response is created by playing a short sound, such as a spark or a starter pistol shot, and recording the acoustical response of the room. The resulting audio sample is applied to the sounds in the game using the convolution operation, making them appear to be playing within that room. Convolution reverb is heavy on the CPU, heavy on memory, and requires you to purchase, find, or record impulse responses, which can be costly or time-consuming. However, once you have good impulse responses, it sounds amazing, convincingly recreating even the complex acoustical environments, and requires little tweaking and tuning.

The game continuously checks which room the audio listener is in and applies the reverb preset assigned to that room to all currently playing 3D sounds. The listener is typically positioned on the player-controlled character's head or on the game camera. Simple to implement and quite effective, this solution has been used in many games, including the Hitman series. However, it is also very rudimentary and has several important drawbacks.

One drawback can be demonstrated with a simple example. Imagine yourself as a game character taking cover in a small concrete bunker and fighting off enemies shooting at you from the hilly countryside. Our simple solution will apply the bunker's reverb to the sound of your enemies' gunfire as well as to the sound of your own weapon. Instead of echoey reverb associated with open, hilly areas, the enemies' shots will have a thudding, metallic ring to them, as if they were fired from inside your concrete bunker. This does not give you as the player correct indication of the sound's environment, and may feel unnatural. In real life, the sound coming from the fields is colored and complemented by reflections and echoes from the hills first, and to a greater extent than by reflections from the walls of the player's bunker.

To address this problem, each sound's reverb should simulate reflections from surfaces that occur as the sound propagates to the listener. Again, that would be very expensive on the CPU. But we can roughly approximate this by finding out which rooms sound traverses before reaching the listener, and chaining reverb presets of these rooms on that sound. Even only applying reverb of the room where the sound emitter is located on each individual sound can, in many cases, give better spatial clues compared to applying the listener room's reverb on all sounds.

Look at the video in Figure 2 for a demonstration of different reverbs and different ways of applying them to sounds in Hitman 2.

Embedded video showcasing in game reverb

Figure 2: "Audio Feature Explained: Reverb" video illustrates the points made in this section.

Using reverb of the locations that the sound propagates from or through poses several challenges:

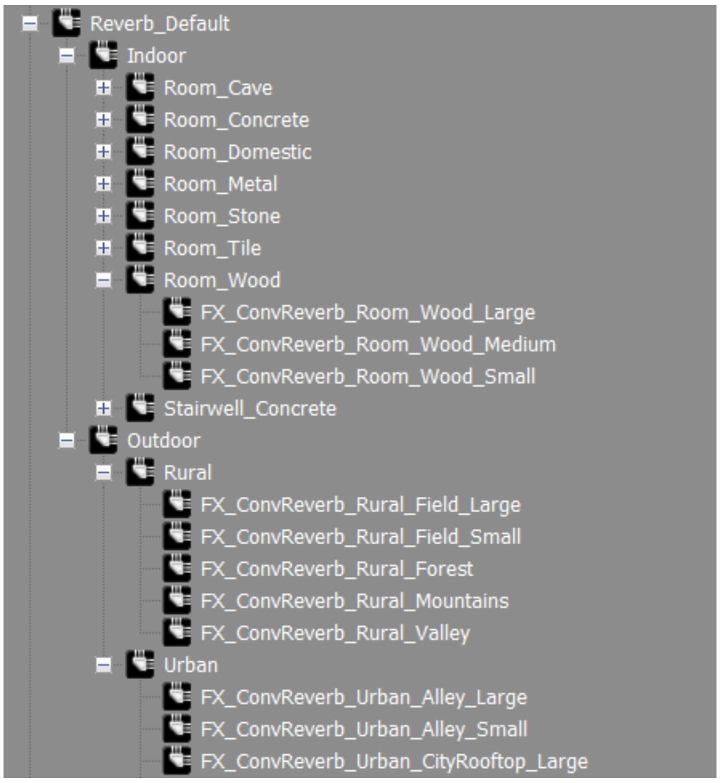
- CPU cost of rendering reverb effects
- Detecting rooms
- Chaining reverbs
- Reverb directionality
- Scaling on different CPUs

Let's look at each one of these in detail.

CPU Cost of Rendering Reverb Effects

Reverb effects tend to be heavy on the CPU, which requires games to minimize the number of effect instances active at the same time. Therefore, reverb effects are generally not applied on individual sounds. Instead, sounds that need a certain reverb effect are routed to a so-called *auxiliary bus* that submixes them and applies the desired effect once. The reverb intensity for each sound is controlled simply by scaling the sound by a certain volume, the so-called *send value*, as it gets submixed into the bus. This is the standard practice in most games. In Wwise, the routing is done on sound emitters using the *SetGameObjectAuxSendValues* API.

Hitman 2 has several dozen reverb busses representing typical environments, as shown in Figure 3. Each room in the game gets assigned one of these busses by sound designers.



```
Reverb_Default
    Indoor
        Room_Cave
        Room_Concrete
        Room_Domestic
        Room_Metal
        Room_Stone
        Room_Tile
        Room_Wood
            FX_ConvReverb_Room_Wood_Large
            FX_ConvReverb_Room_Wood_Medium
            FX_ConvReverb_Room_Wood_Small
        Stairwell_Concrete
    Outdoor
        Rural
            FX_ConvReverb_Rural_Field_Large
            FX_ConvReverb_Rural_Field_Small
            FX_ConvReverb_Rural_Forest
            FX_ConvReverb_Rural_Mountains
            FX_ConvReverb_Rural_Valley
        Urban
            FX_ConvReverb_Urban_Alley_Large
            FX_ConvReverb_Urban_Alley_Small
            FX_ConvReverb_Urban_CityRooftop_Large
```

Figure 1: Reverb bus hierarchy in our Wwise* project.

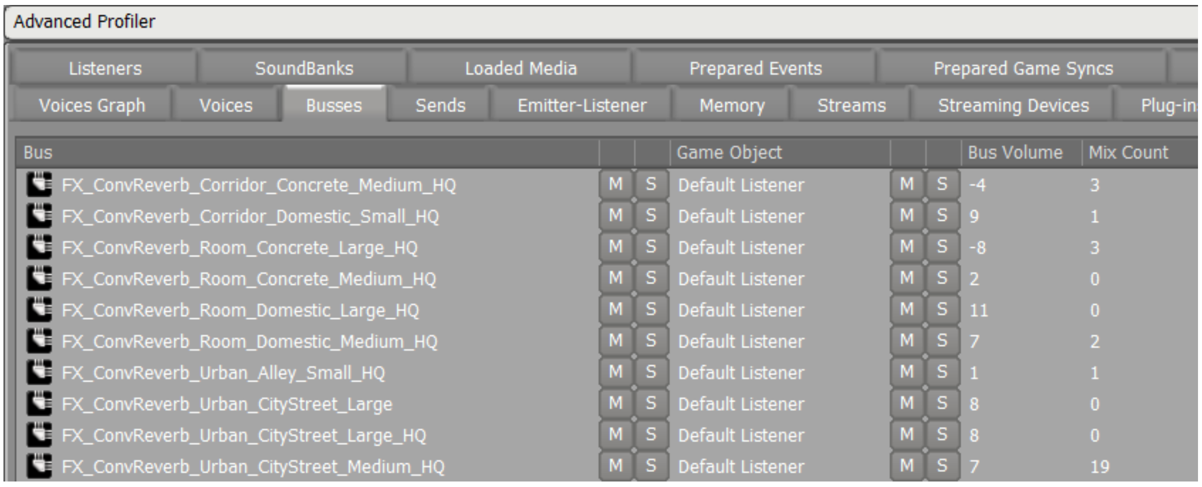
An effect on a bus only gets rendered when that bus is active or, in other words, when there are playing sounds being routed to it. Busses with no playing sounds are dormant and their effects generally do not use the CPU. The exception to that is when all sounds routed to the bus have ended but the bus remains active, to finish playing the tail generated by its effects, such as reverb.

CPU Footprint of the Previous Solution

The previous solution is inexpensive in terms of CPU usage. Routing all sounds to the listener room's bus means that we render only one reverb effect. More than one simultaneously active bus occurs when the listener stands near portals that connect rooms (for example, doors and other openings). In that case, we need to route sounds not only to the listener room's bus, but also to the busses of rooms on the other side of the nearby portals. This ensures smooth reverb transition when the listener moves from one room to another. Even so, the active bus count never exceeds two or three, which makes CPU cost of the effects used less relevant.

CPU Footprint of Our New Solution

It is a different picture once we start routing sounds to the busses of rooms where they are located. Given that the maximum number of simultaneously playing sounds (voice cap) in Hitman 2 is 64, we could in theory end up with as many as 64 active reverb busses, if each sound is in its own room with its own unique bus. While in practice this worst-case scenario is extremely unlikely, the active reverb bus count is nonetheless higher than with the old solution, averaging about 8 to 10 at a time. For example, look at the screenshot from Wwise Advanced Profiler in Figure 4 that shows active reverb busses in a typical level as well as the number of sounds that get mixed into each bus (Mix Count column).


Advanced Profiler
Listeners | SoundBanks | Loaded Media | Prepared Events | Prepared Game Syncs
Voices Graph | Voices | Busses | Sends | Emitter-Listener | Memory | Streams | Streaming Devices | Plug-in

Bus			Game Object			Bus Volume	Mix Count
FX_ConvReverb_Corridor_Concrete_Medium_HQ	M	S	Default Listener	M	S	-4	3
FX_ConvReverb_Corridor_Domestic_Small_HQ	M	S	Default Listener	M	S	9	1
FX_ConvReverb_Room_Concrete_Large_HQ	M	S	Default Listener	M	S	-8	3
FX_ConvReverb_Room_Concrete_Medium_HQ	M	S	Default Listener	M	S	2	0
FX_ConvReverb_Room_Domestic_Large_HQ	M	S	Default Listener	M	S	11	0
FX_ConvReverb_Room_Domestic_Medium_HQ	M	S	Default Listener	M	S	7	2
FX_ConvReverb_Urban_Alley_Small_HQ	M	S	Default Listener	M	S	1	1
FX_ConvReverb_Urban_CityStreet_Large	M	S	Default Listener	M	S	8	0
FX_ConvReverb_Urban_CityStreet_Large_HQ	M	S	Default Listener	M	S	8	0
FX_ConvReverb_Urban_CityStreet_Medium_HQ	M	S	Default Listener	M	S	7	19

Figure 2: Active reverb busses. Note that some busses are active even though their Mix Count is 0. They are playing reverb tail of sounds that already ended.

The 8 to 10 reverb instances may not seem like a huge difference from the 2 to 3 of the old solution. However, we use convolution reverb, which has a high CPU cost that scales linearly with the number of reverb instances. Switching to the new system thus increases the total audio frame rendering time by over 50 percent, depending on the reverb settings. This in turn may lead to stuttering if rendering is not done before the hardware runs out of sound data to play, or at least to a negative impact on the frames per second (fps).

Parallelizing Audio Rendering

The good news is that reverb busses have no dependencies on one another, and their effects write to the bus's own submix buffer. Therefore, they are perfect candidates for parallel execution. This is where modern multicore CPUs come into play, because rendering reverb effects concurrently on different hardware threads can reduce or even completely cancel the negative impact of the increased number of reverb instances.

Because we use Wwise, we get this for free. Task-based audio rendering implemented by Intel engineers is included in the standard Wwise software development kit (SDK) since version 2018.1. It breaks down the audio rendering pipeline into sets of granular tasks that can be done in parallel, and asks the game to execute them. We feed these tasks to our own task scheduler, which executes them on the available worker threads as shown in Figure 5. Games that use Wwise but do not have a task scheduler may benefit from the sample implementation provided in the SDK (see TaskScheduler sample). It takes little work to start using it, and the improvement in audio frame rendering times is very tangible.

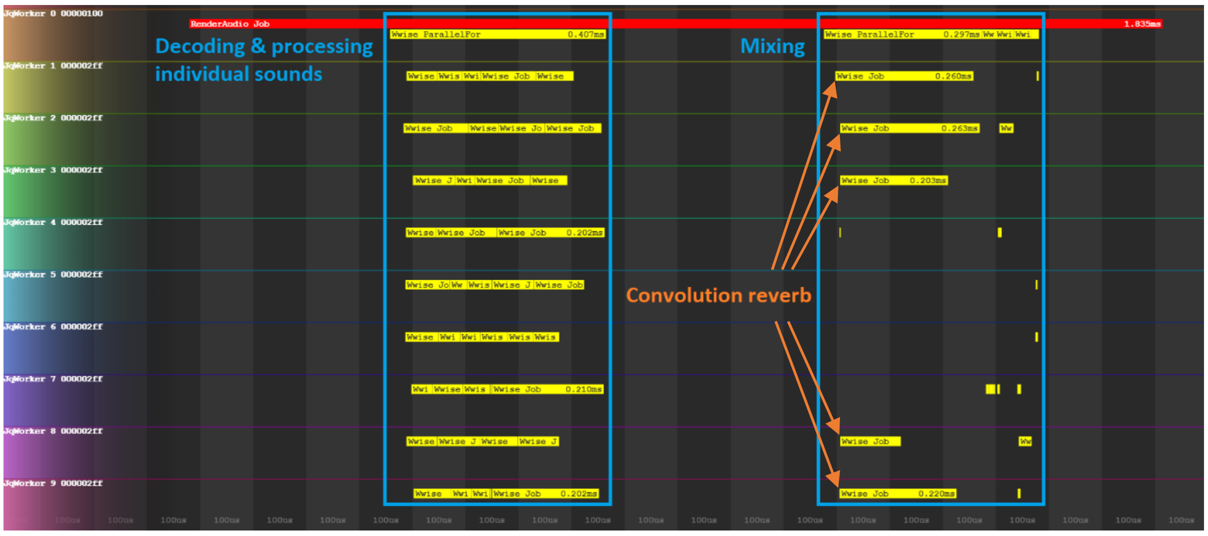


Figure 3: This screenshot of the Glacier profiler shows parallelized rendering of a typical audio frame in Hitman* 2. Horizontal lines entitled “JqWorker” are worker threads, whereas red and yellow items represent individual tasks (“jobs” in Glacier terminology) executed on these threads. The view is filtered to show only audio tasks.

Performance Comparison

Let’s look at performance with and without parallelized audio rendering. For this comparison, we measured average audio frame rendering time when idling at the location shown in Figure 6.

Figure 4: The spot in the Miami level of Hitman* 2 used for performance measurements.

In the first test case the old reverb system is enabled. The test location is in the vicinity of a portal, so there are two active reverbs. For the second test case we switch to the new solution, which results in eight active reverbs. And finally, we demonstrate what happens when the new system is combined with the highest quality reverb presets that sound best, but also are the slowest.

In all cases, Wwise Convolution Reverb is the effect used. For those familiar with this plugin, the presets used in the first two tests have mono impulse responses (IRs) and a threshold of -50 dB, whereas for the last test switched to presets with stereo IRs and a threshold of -144 dB. The length of IRs used in regular and high-quality presets is identical and varies from one to three seconds for different presets.

Figure 7 shows results of these tests on a PC with six-core CPU.

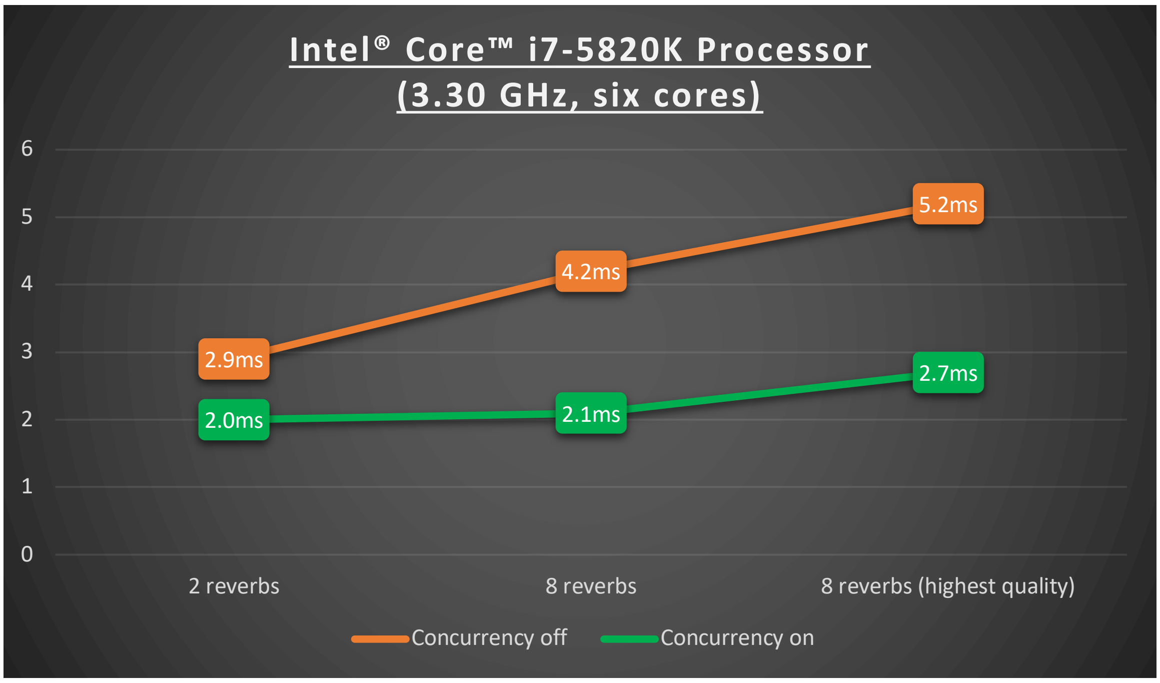


Figure 5: Audio frame rendering times with and without parallelization.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information, see Performance Benchmark Test Disclosure.

Performance results are based on testing by IO Interactive as of May 17th, 2019 and may not reflect all publicly available security updates.

Configuration: Intel® Core™ i7-5820K processor @ 3.30 GHz, NVIDIA GeForce GTX 970, 32 GB RAM, Windows 10 (build 1809)

As you can see, using parallelized audio rendering results in a 1.45x improvement in the first test, 2x in the second, and 1.9x in the third, compared to the non-parallelized version. More importantly, it improves scaling, as the number of reverb instances and the CPU cost of each instance increase.

On the test PC, each task took on average 60 microseconds, which is a rather good task granularity. Convolution reverb tasks were the notable exception, taking as much as 1 millisecond (ms) to render effects with the highest quality presets. However, even in the most expensive test case with eight high-quality reverbs, the total time per game frame spent executing audio tasks on all cores averages to only 17 percent of the frame time of a single core. This means that the audio tasks take up a relatively small amount of time on each core, leaving most CPU resources for other game tasks.

To get a very rough idea of how the same tests would fare on Xbox One* and PlayStation 4*, multiply times from Figure 7 by four.

Figure 8 compares performance gains achieved by parallelization on CPUs with different numbers of cores. It clearly demonstrates that concurrent execution yields an improvement even on older CPUs with four and six cores, with modern eight-core CPUs enjoying the biggest gains.

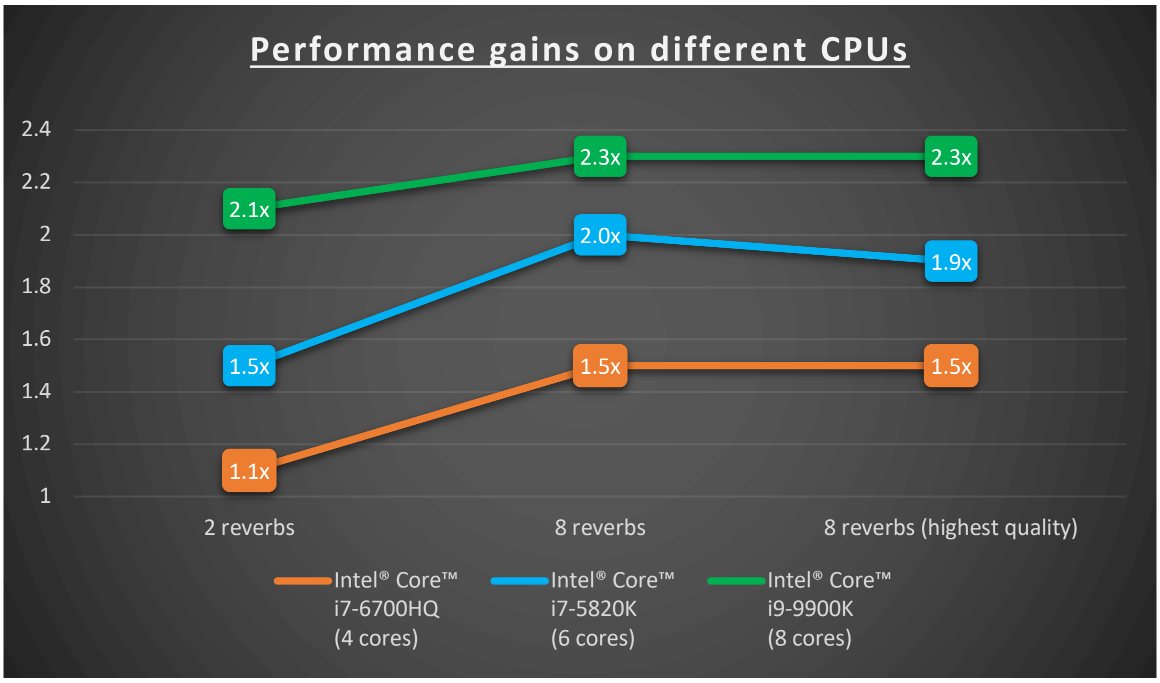


Figure 6: Performance gains from concurrency on Intel® Core™ processors with different numbers of cores. For each test case and CPU, the gain factor is calculated by dividing the synchronous audio frame rendering time by the parallelized rendering time.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information, see Performance Benchmark Test Disclosure.

Performance results are based on testing by IO Interactive as of May 17th, 2019 and may not reflect all publicly available security updates.

Configurations:

- Intel® Core™ i9-9900K processor @ 3.60 GHz, NVIDIA GeForce* GTX 1070, 16 GB RAM, Windows 10 (build 1809)
- Intel® Core™ i7-5820K processor @ 3.30 GHz, NVIDIA GeForce GTX 970, 32 GB RAM, Windows 10 (build 1809)
- Intel® Core™ i7-6700HQ processor @ 2.60 GHz, NVIDIA GeForce GTX 970M, 32 GB RAM, Windows 10 (build 1809)

Detecting Rooms

To determine which reverb bus a sound should be fed to, we need to find the room where that sound is located. For this purpose, we store rooms in a spatial subdivision data structure that allows fast lookup from a point (sound emitter position). This algorithm is described in detail in the Game Developers Conference Europe (GDCE) 2015 talk, *Sound Propagation In Hitman*.

The implementation is straightforward; the only caveat to watch out for is making sure we pick the right environment among several that are nested or overlapping. This can be facilitated by assigning priority to rooms. Outermost rooms should have the lowest priority, whereas innermost rooms have the highest. The priority should be calculated automatically, with an option to be overridden manually in the editor for fine tuning.

Another thing to think about is preventing an abrupt change of reverb when a sound emitter moves from one room to another. This is done by taking nearby portals and routing the sound emitter to the busses of rooms on the other side of these portals. The send value in this case is a factor of distance between the sound emitter and the closest opening that leads into a different room.

To implement this, we need an efficient way to get a list of portals within a certain distance of a point. Therefore, we store geometry representing portals in the same type of spatial subdivision structure as we use for rooms.

Querying for rooms and nearby portals and routing to the appropriate busses needs to be done for all sounds on every frame. We can avoid doing it for sounds that didn't change position, assuming the geometry didn't change. But there can still be a handful of these operations to perform, and while they are relatively lightweight, together they may add up to undesirable overhead on the main thread. It is therefore a good idea to offload them to the worker threads.

One approach is to subdivide all queries into subsets, with each subset being processed in its own task. The number of tasks should correspond to the number of worker threads, so choose the subset size with that in mind. Schedule the tasks right after the game logic update is done, so that no new emitters may be spawned while we execute queries. They should complete before we begin processing the audio command queue, or the call to *RenderAudio* if you use Wwise.

Yet an even better approach is to make the entire sound emitter update routine an individual task and execute these tasks in parallel on the worker threads. *The update routine* here is a collective name for any calculations you do on the main thread on each frame for each playing sound emitter. It may include occlusion, diffraction, volumetric, Doppler, and reverb calculations.

Chaining Reverbs

Applying reverbs of the rooms that the sound travels through on its way to the listener requires the rooms' busses to be chained. Let's say you have some sounds in room A and the listener in room C. Rooms A and C are connected through room B. We begin by routing the sounds to room A's bus. That bus is routed to room B's bus, and the latter is routed to room C's bus. As a result, the sound is affected by reverb from all three rooms that it passes through.

Performance note: Regardless of the number of sounds in all three rooms, the active bus count would still be three. But if all playing sounds are in different rooms or propagate through many different rooms, then you quickly end up with too many active busses. In that case, you need to implement logic to reduce their count; for example, by choosing not to route sound to a bus if this is not expected to have a significant impact on the overall mix.

The propagation system described in the aforementioned GDCE talk can be extended to keep track of multiple propagation paths and rooms in each propagation path. Once you have that information, it is rather trivial to set up bus routing.

Bus Routing Setup in Wwise*

A blog post by Audiokinetic entitled "Working With the New 3D-Bus Architecture in Wwise 2017.1: Simulating an Audio Surveillance System" explains the setup in detail. Even though its case in point is implementing an audio surveillance system, the same principles apply for setting up chaining reverbs. You need to enable Listener Relative Routing (known as Enable Positioning before Wwise 2018.1) on the reverb busses, register a sound emitter for each room, and then use the *SetListeners* API to route sound from one room emitter to another. Finally, when setting up reverb bus routing for a regular sound emitter using the *SetGameObjectAuxSendValues* API, set *AkAuxSendValue::listenerID* to the emitter ID of the room where that regular emitter is located.

Parallelizing Propagation System

The original propagation system written for Hitman (2016) has only been used for calculating occlusion and diffraction; hence it only concerned itself with finding the best propagation path from any room into the listener's room. As a result, updating the propagation system was cheap, taking about 0.1 ms per frame, and we used to do it on the main thread. However, tracking multiple paths through which the sound can reach the listener is more expensive, so it made a lot of sense to offload it to the worker threads.

The usual consideration when doing this is to avoid waiting for the update to complete on any other thread at any time. We achieve this using two simple tricks:

- *Timing:* The propagation system update is scheduled early in the frame, after updating the camera and the game character position. That's when the listener position becomes known. This gives the propagation task plenty of time to complete before propagation data is needed by the audio system update at the end of the frame.
- *Double-buffering propagation data:* New sounds spawned by the game loop may query the propagation system while it is being updated, therefore we cache propagation data from the last frame and serve it throughout the current one. Propagation data is swapped at the end of the frame, before the audio system update.

Reverb Directionality

An important problem that pertains to the old single-reverb solution, but even more so to the new one, is that most reverb effects used in games do not preserve directionality of sounds because they are single channel. This is done for performance reasons. The consequence is that the reverb is audible in all speakers, no matter the direction that the sound comes from. For example, assume a non-player character (NPC) fired a gun to the left of your game character. While the direct sound will clearly be audible from the left, its reverb will seem to emanate from all sides. This is fine if both the sound and the listener are in the same enclosed space, but in practically any other case we want to control directionality and spread of the reverb. This is especially true when many reverbs can be active at the same time. While lack of directionality can be tolerated with one or two reverbs, having many omnidirectional instances may result in a confusing and bad-sounding mix.

Panning Reverb Busses

An easy way to solve the directionality problem is to treat the reverb output for each room as a standalone 3D sound and give it direction and volume by using panning and spread.

The steps to implement this in Wwise are practically the same as described in the Bus Routing Setup in Wwise section. You need to enable "Listener Relative Routing" on the reverb bus and associate that bus with the room emitter by assigning that room's emitter ID to *AkAuxSendValue::listenerID* when you set up reverb routing for regular emitters located in the room. Now you can pan the reverb by positioning the room emitter at the center of the room or on portals leading into the listener's room (more realistic but harder to implement). You can even implement custom mixing using the *RegisterBusVolumeCallback* API, which is useful since it is the only way in Wwise to control spread based on something other than the distance between the emitter and the listener.

The obvious drawback to this method is that the reverb of the listener's room cannot be panned this way. This may be a problem if the room represents a large open space. You can work around that by programmatically breaking up such large spaces into several parts, creating a sound emitter for each part and routing sounds to their respective parts. The same may have to be done for large non-listener rooms as well. Breaking up geometry can be offline as part the geometry generation process.

The other drawback is performance related. By enabling positioning on reverb busses and using a sound emitter per room, you are effectively creating an active bus instance for each room. Whereas before, two rooms using the same reverb bus would share the active bus instance, spawning only one reverb effect, now they would each have their own bus instance. The same applies to the geometry you break up into smaller parts as described above—each part will spawn a bus instance as well. This may or may not be a problem depending on your geometry.

Using Multichannel Reverb Effects

Another way to preserve reverb directionality is to use multichannel reverb. Four channels should be enough unless supporting verticality is required. Each sound gets mixed into the reverb bus channels with respect to its positioning relative to the listener. Once all sounds have been submixed, the reverb effect gets rendered separately on each channel, except for the channels that do not have audible content for this frame. Then, the reverb bus channels are upmixed into the final output audio bus, whose channel configuration corresponds to the player's actual speaker setup (stereo, 5.1, 7.1, and so on).

The reverb cost is multiplied by the number of channels when using this approach. Therefore, it can be prohibitively expensive to use this effect on all active reverb busses. The optimal approach is to use mono reverb effect and reverb bus panning for non-listener rooms and multichannel reverb effect for the listener room. When the listener approaches a portal leading into an adjoining room, the mono reverb of that room should be crossfaded with its multichannel version to ensure smooth transition.

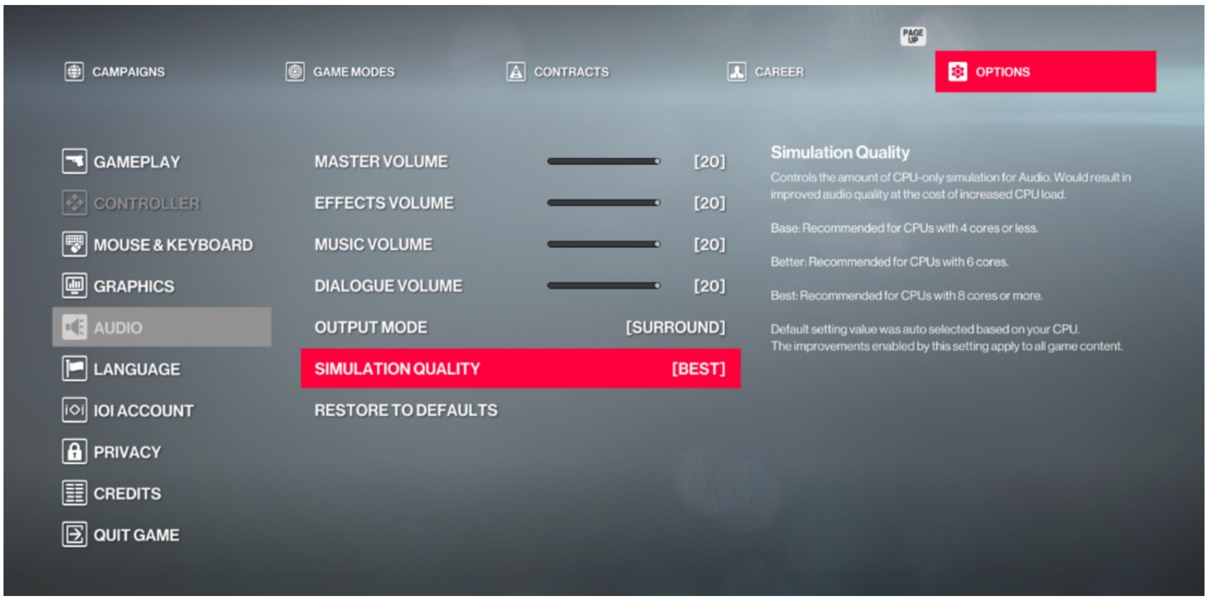
Wwise does not have support for multichannel reverb effects. Wwise Convolution Reverb has a *Filter* mode, which functions in a way close to the logic described above. However, it is not flexible enough with regard to various channel configurations and for this reason cannot be used in production. Nor is this mode meant to be used for simulating reverberation. Therefore, using multichannel reverb is an option only for those willing to develop their own convolution reverb plugin. When implementing such a plugin, try to take advantage of parallelization—rendering the effect on each channel should be an individual task executed on a worker thread. These tasks have no mutual dependencies during execution.

Scaling on Different CPUs

In Hitman 2, we kept support for the old system and have shipped with it on consoles. On PCs, however, the player can choose between three quality levels, collectively referred to as *audio simulation quality*:

- *Base*: No audio enhancements. Default on CPUs with four physical cores or fewer.
- *Better*: New reverb system. Default on CPUs with more than four physical cores.
- *Best*: Same as *better*, but with highest quality reverb settings. Default on CPUs with more than six physical cores.

The player can control audio simulation quality in the Audio Options as shown in Figure 9.



MASTER VOLUME [20]
EFFECTS VOLUME [20]
MUSIC VOLUME [20]
DIALOGUE VOLUME [20]
OUTPUT MODE [SURROUND]
SIMULATION QUALITY [BEST]

Simulation Quality

Controls the amount of CPU-only simulation for Audio. Would result in improved audio quality at the cost of increased CPU load.

Base: Recommended for CPUs with 4 cores or less.

Better: Recommended for CPUs with 6 cores.

Best: Recommended for CPUs with 8 cores or more.

Default setting value was auto selected based on your CPU.
The improvements enabled by this setting apply to all game content.

Figure 7: Audio simulation quality setting.

Wwise Implementation

Supporting different levels of quality for reverb effects in Wwise can be done by duplicating the reverb bus hierarchy as shown in Figure 10, and using better effects for high-quality busses. The code automatically picks the correct bus (standard or HQ) based on the simulation quality value.

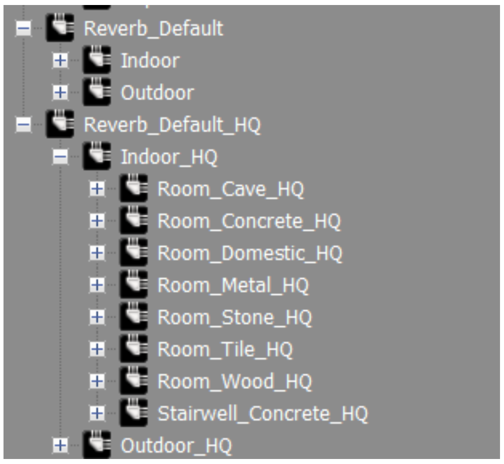
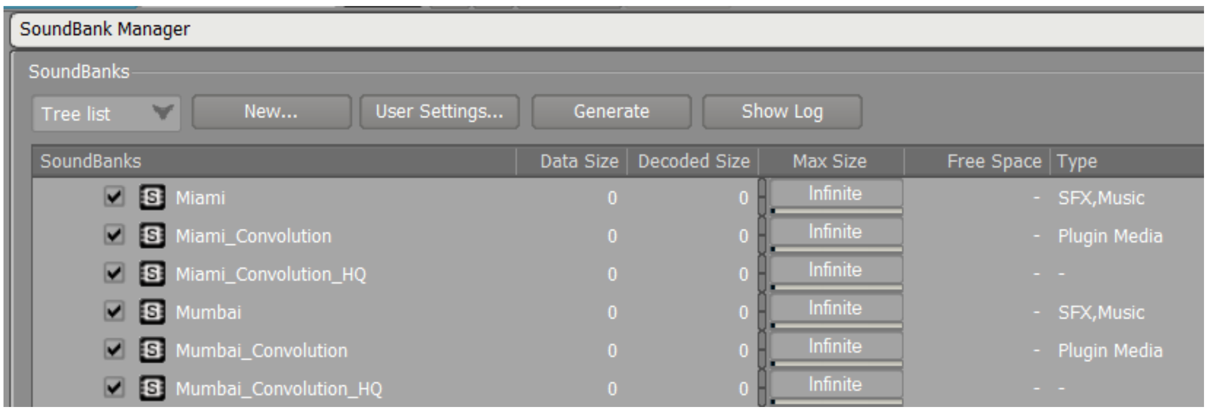


Figure 8: High-quality busses mirror the regular reverb bus hierarchy.

Another approach, which may be easier to maintain, is to keep one hierarchy and add both effects to each reverb bus, and bypass one of the effects at runtime using a global Real Time Parameter Control (RTPC) set from code.

Whichever way you choose, you need to store the effects' media in different sound banks and load the correct one based on the simulation quality value to avoid unnecessary memory usage overhead. We use a different convolution reverb sound bank for each level, as shown in Figure 11.



SoundBank Manager

SoundBanks

SoundBanks	Data Size	Decoded Size	Max Size	Free Space	Type
✓ Miami	0	0	Infinite	-	SFX,Music
✓ Miami_Convolution	0	0	Infinite	-	Plugin Media
✓ Miami_Convolution_HQ	0	0	Infinite	-	-
✓ Mumbai	0	0	Infinite	-	SFX,Music
✓ Mumbai_Convolution	0	0	Infinite	-	Plugin Media
✓ Mumbai_Convolution_HQ	0	0	Infinite	-	-

Figure 9: Sound bank structure in Hitman* 2.

Conclusion

This article shows how utilizing multiple cores on modern CPUs can open possibilities for improving game audio that have previously been inaccessible due to the high computational cost. Changes to the reverb system described here have been noticed and enjoyed by players of Hitman 2. Better parallelization in the audio system has freed up CPU resources that have been put to good use by the audio team and other departments.

Yet this example is still affected by the limitations of the lower end of the spectrum of multicore CPUs. The widespread adoption of systems with eight+ cores, the conscious effort to parallelize calculations by the game developers and leveraging vectorization (for example, Intel® Advanced Vector Extensions 2 (Intel® AVX2) or Intel® Advanced Vector Extensions 512 (Intel® AVX-512)) by the audio middleware producers may bring even more detail and immersion to the sound in games.

About the Author

Stepan Boev is the lead audio programmer at IO Interactive. Over the course of his 14 years in the video games industry, he has worked on the *Hitman*, *Far Cry** and *World in Conflict** franchises. He is passionate about aesthetics, immersion, and detail in game audio as well as solving complex programming problems and performance optimization.

References

Hitman 2 official website.

IO Interactive official website

Game Developers Conference Europe (GDCE) 2015 talk, *Sound Propagation In Hitman*.

Working With the New 3D-Bus Architecture in Wwise 2017.1: Simulating an Audio Surveillance System

Notices