**CO-PERF: A COLLABORATIVE PERFORMANCE BENCHMARKING APPLICATION**

Eric Preisz 1, Shawn Stafford 1, Ben Garney 2, Jeremiah Blanchard 1, Cromwell Enage 1

**ABSTRACT**

This paper describes how Co-Perf, an open source benchmarking system, will provide capabilities beyond those of existing game benchmarking tools. We present a system that supports not only the game consumer, but also the industry developer and academia.

Full Sail Institute for Research and Entertainment is launching a collaborative performance benchmarking system named Co-Perf. This benchmarking system is a collection of lightweight CPU and GPU benchmarks we call “micro benchmarks” that collect data and drive a reporting system that coalesces results for use in determining optimization strategies and consumer purchasing decisions.

Benchmarking data, when considering the permutations of drivers, hardware, and configuration settings, requires a large set of test machines that are representative of current hardware install bases; Co-Perf addresses this need by appealing to thousands of game consumers by providing free information about their current machines and a speculative view of their machines based on potential computer upgrades.

**KEY WORDS**

Benchmarks, Game Development, CPU, GPU, Game Performance

**1 Introduction**

Co-Perf is a system of open source benchmarks and a web-based user interface that generates reports for game consumers while also providing detailed data for industry and academia.

Users can execute the benchmark suite by either downloading the current binary release or by compiling the source code. After running, the executable generates a report. The report unlocks a website via web service.

Users create or receive an id that is allows valid access to the reporting for a specified duration. The report implements security measures that reduce users’ abilities to tamper with data.

Existing commercial benchmarks are not open source and are vulnerable to implementations that may skew final results and conclusions. Co-Perf addresses this issue by providing a public, open source view of every benchmark.

Current benchmarks for games typically use two approaches. The first format uses a repeatable, high-end, game like applications that generates scores based on the performance in areas of the benchmark. The second approach, provides detailed information on key statistics such as triangles per second. Co-Perf uses focuses on the latter, with the option of including the former if necessary.

Co-Perf benchmarks will be valuable to industry and academia. By providing developers with the source code and the metrics data they are able to adjust their coding practices to those that show the best practices and historical trends for their target hardware. The academic domain can incorporate the metrics into classrooms for use in teaching game programming best practices.

Co-Perf benchmarks are valuable to game hardware consumers. In order to generate a wide array of data, we plan to attract game consumers to Co-Perf. Co-Perf will allow users to view statistics about their machines and other machines that are similar to their configurations. Game consumers will be able perform a cost-benefit analysis on individual hardware components such as ram upgrades and GPU upgrades.

By showing consumers which benchmarks did not perform well we can give insight into what game features may cause performance bottlenecks. For example, if a particular configuration is lacking raster operations bandwidth, we could suggest to a user that this card may not perform well in games with high amounts of overdraw caused from in-game features such as dense alpha enabled grass or particle systems.

**2 Related Work**

*Futuremark <http://www.futuremark.com/>*

Futuremark Corporation specializes in 3D and PC benchmarking software. The version that we tried out, 3D Mark 06, runs up to six game engine demos as well as several individual feature tests in order to measure the computer’s CPU, SM2.0, and HDR/SM3.0 capabilities; you can configure which tests should or should not run. During each test, 3D Mark 06 displays the current frame rate so that a viewer could see which visual effects affect or do not affect it; after completing all the user-requested tests, the software tallies the average frame rate and other performance scores for each test. Current versions of the software can execute only on Windows Vista with DirectX 10 support.

*AquaMark 3*

*<http://www.softpedia.com/get/System/Benchmarks/AquaMark.shtml/>*

Here’s another offering from 2006 that uses a full-fledged game engine to benchmark gaming performance. Its USP is that it also tests the features of PS2.0 from DirectX 9 while staying compatible with DirectX 7 and DirectX 8.

*ShaderMark v2.1*

*<http://www.shadermark.com/>*

This German import targets Microsoft’s HLSL for benchmarking. The software has not been updated since build 130, June 2005.

*Rightmark <http://www.rightmark.org/>*

This software suite benchmarks not only the graphics card and the CPU, but also RAM and audio cards. Unlike the previous offerings, most of its tests are open-source, with the programs written in C++. However, the Direct3D component does not have its source code available, and the web page has not been updated since 2004.

**3 Challenges & Solutions**

As with any program, development challenges exist. In this section, we cover some of the known challenges and current solutions and/or considerations.

*User Software Configurations*

Users are able to configure settings that will unfairly skew test results. We have two philosophies concerning this issue. 1) Gather the information since this is representative game consumers. 2) Detect these settings and exclude the data from the database. At this time we are requesting advice from industry.

*Small Sample Sets*

Initial thoughts require us to have 200-300 machines of a given configuration to reduce errors and increase our statistical power. We estimate that there are thousands of relevant computer configurations. Custom systems with a low reporting rate due to obscure components could utilize a taxonomic view with main system components (CPU, GPU, motherboard, memory) positioned as “kingdom” components at the top of the benchmarking hierarchy. Novel solutions should help address the issue of attracting game consumers to the reporting web site.

*Benchmarking in a Lab vs. the Field - Internal vs. External Validity*

Internal validity of benchmarking performance scores is often controlled by the use of a clean lab space and the educated computer technician. Unfortunately, even in these carefully controlled settings we open ourselves to the threats posed by the use of a very limited sample size (typically one system for each configuration). The threat of type 1 error that is posed by using a sample size of one is often overlooked by computer technicians who build the systems and by those enthusiast who flock to their websites. A set of one does not allow for an analysis of the standard deviation and its impact on the often minor distance between two points (representing two competing products) on a performance curve. In plain words we might ask, “Is this performance point robust enough to allot for the purchase of an upgraded component?” The reverse situation also poses a variety of threats. Naturalistic benchmarking of home systems allows for multiple systems thus increasing the statistical power and reducing the threat of type 1 error while introducing variability in scores through uncontrolled system design, power quality, and heat. This could lead to type II error - “The data is too random and therefore I should not upgrade.” We can help control for type II error by increasing sample size and providing users with the knowledge they need about data points and error bars so the consumer can make an informed decision. We can comfortably state that control for type 1 error in the first scenario is, for all practical purposes, financially restrictive – and thus effectively unachievable!

*Isolating Hardware Components*

Some tests require us to isolate hardware components. This can be very difficult, especially on the GPU. A set of common benchmark functions will reduce this complexity. For example, a vertex limiter is part of the benchmark system utilities. This limiter will reduce the frame rate using static vertex buffers of degenerate triangles until the frame rate is 60 Hz. The benchmark, using the limiter, will then increase workload until performance reaches 30 Hz. For these benchmarks, the performance between 60 Hz and 30 Hz is the most valuable.

*Data Validity*

Users will create or receive a unique identifier to use Co-Perf. A list IDs from trusted partners will enable the mining of trusted data. This trusted data will act as a comparison against all data for purposes of validity.

*Benchmark Validity*

We believe that an open source environment will add in making the benchmarks valid. It is our hope that industry and academia will provide feedback that will ensure repeatable, reliable benchmarking.

*Many APIs and Compilers*

There are many graphics APIs we can benchmark. Near term graphics APIs are limited to Dx9, Dx10, and OpenGL. At this time, we are currently developing Dx9 GPU tests; Dx10 and OpenGL tests will follow. Dx9 tests are implemented by using the DXUT which facilitates Dx9 or Dx10 benchmark development.

At this time, it is unclear whether or not certain compilers should be mandated. This data could be serialized in the data output using preprocessor defines.

*Versioning*

All data will contain version numbers. If a release requires it, a new version number will be generated. Versioning also allows us to filter data on the reporting web-site. Version numbers are per release of a given benchmark.

V-Sync

Applications that enable v-sync will execute and generate data, but will be consider as standard data included in typical reports. We will warn the user that forcing v-sync will drastically affect the results of their benchmark and data validity.

**4 Implementation**

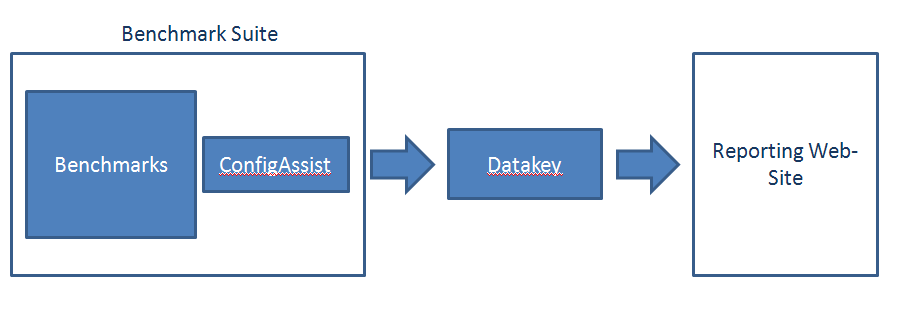


Figure 1 The benchmark suite generates the data key for the reporting web-site.

The benchmark system contains many micro benchmarks and the ConfigManager utility. ConfigManager gathers information about the machine configuration.

*Test Harness*

The test harness is the foundation for all benchmarks. It contains a simple API to reduce redundancy between benchmarks.

At this time, we intended to collect min, max, mean, standard deviation, and standard error for each iteration of a given benchmark.

The test harness will execute each benchmark as its own process. In doing so we reduce contamination of hardware resources across benchmark runs.

*Utilities*

A set of useful common functions will enable benchmark developers. These common functions include:

*Cache clearing* – this function allocates a large piece of memory, steps through the memory, and deletes it.

*Vertex Limiter* – the vertex limiter allocates degenerate triangles until the frame rate reaches 60 Hz.

*Pixel Limiter* – the application uses very expensive pixel shaders until the performance is 60 Hz.

*Framebuffer bandwidth limiter* – the application renders full screen quad with simple shader and no texture until the frame rate is 60 Hz.

*Full Screen Quad Renderer* – the full screen quad renderer draws a quad with or without a texture and with or without a complex shader.

*Mersenne twister* – a utility for generating random numbers.

*File Format* – at this time we are opting for a binary or text version of XML.

*Development Environment*

*Work Sharing* – we are using Google code for the open source repository.

*Mailing List* – we also have a Google groups account for managing group discussions that are not appropriate for the Google code wiki. We encourage industry to join our mailing list.

*Reporting* – The reporting web-site will preferably use a flash or ajax based front-end. The landing page should be very friendly for game consumers since they are the less likely to search for more detailed information.

Industry and academia, as well as the consumer, will be able to drill down into finer grain detail. The lowest level detail includes raw data from the benchmarking system and/or custom data searching.

**5 Benchmarks**

At this time, the following benchmarks are in development, proto-type, or consideration. Existing tests focus on current fixed hardware; future tests will evolve to support configurable rending pipelines of future rendering systems.

*GPU Tests*

**Vertex Buffer Size** -A single vertex buffer will have an optimal size. Build a vertex buffer of varying size from small to large and determine, via framerate, what the optimal size is for the hardware. Start with the vertex limiter so that the program begins at 60 hertz. The test is over when the fps reaches 30 fps.

**Vertex Alignment** - Start by implementing the vertex limiter. Using different permutations of vertex types, increase buffer size until the fps is 30 Hz. Report the total size of the vertex (number of bytes\*sizof(vertexformat)). The assumption is that vertex types that are aligned to the pre-tnl cache boundaries of the vertex fetching system should be faster than those that are not. Vertex buffers must be static as to reduce graphics bus bandwidth usage.

**Overdraw Performance** - use a full screen quad and a small frambuffer to gather the frame rate at overdraw values of 0,1,2,3,4…x. Use the fullscreen quad draw to bring the frame rate down to 60. Once there, continue to add fullsreen quads until the frame rate is at 30 Hz. Record the fps and number of quads.

**Draw Call Performance** - too many draw calls with renderstate changes cause CPU overhead. X number of draw calls using the same render state, on some drivers, is the same as one draw call. The drivers, in this case, are batching the render states for you. Determine how many different render state changes the driver can handle in a loosely sorted list of states. Create any GPU limiter to bring the frame rate to 60Hz. Using a single vertex buffer, implement a loop to draw the buffer in increments of 100. Now write the program so that it changes render states between draw calls. Change the first render state every other call and measure the frame rate. Change the second render state every third call. Continue this process until the framerate is below 30.

**Frequency of Updating a Static Vertex Buffer** - It has been suggested that some drivers will change a static vertex buffer that is locked more often to a dynamic buffer that utilizes non-local video memory. Use the vertex limiter to create a scene that executes at 60 Hz. Create an fill a vertex buffer until the framerate is 30 Hz. Lock the buffer at intervals other than every frame and track the framerate delta.

**Vertex Shader Flow of Control**- Using the vertex limiter, experiment with different types of vertex shader branches to determine performance. Chart the differences between:

1) If then else statements with random conditions

2) If then else statements without predictable conditions

3) If then else statements with semi-random conditions (interleave true and false on intervals of 31, 32, and 33.)

4) Loops with random conditions

5) Loops with reliable conditions

**Texture Filtering Performance** - Using the texture limiter, create an application that is texture bound. Use a simple shader to test out all types of filtering performance. Use a DXT1 texture as your test texture.

*CPU Tests*

**CPU Stream Count** - Hardware prefetching hides the latency of memory fetching for our CPUs. Hardware prefetching can track more than one "stream" of memory. Create a struct with at least 32 members in a Structure of Arrays (SOA) format. Array size should be 4 megs. In your first test, linearly access memory in the first member. In the second test, iterate through the first and second member, but only iterate halfway through the entire array. The assumption is that the CPU can track multiple streams of memory for hardware prefetching. This test attempts to plot increasing numbers of streams without increasing the total amount of memory streamed.

**Datatypes and Operators** - This test iterates through a series of multiply, divide, minus, and plus operators for different datatypes. To reduce unfair compilation optimizations, perform these tests on series of linear arrays populated with random data of the appropriate types.

**Memory Access Patterns** – Memory systems hide latency by speculative hardeware such as automatic hardware prefetching. This test measures different access patterns in memory. Memory traversals on integer datatypes will be random, linear, and strides of access. (note: this test is already implemented)

**Multi-thread linear access** - perform linear access on an array using a linear memory access pattern. Collect data for [1..64] threads.

**Single vs. multi-thread random** access - perform the same test as the above test replacing linear access with random access.

**6 Conclusions**

The Co-Perf system aims to address the issue of benchmarking through a more distributed (and robust) method than individual component testing in laboratory environments. Its goal is to improve the usefulness of such benchmark data as it applies to the software industry, scientific research, academia, and finally software and hardware consumers.

While such a system presents considerable challenges in the implementation due to sample size, quality, and tampering, we believe that our approaches to these issues (hierarchical arrangement, error tracking, and data validation) will help mitigate problems. In addition, we are seeking feedback from the industry regarding both our approach to the system in general and to specific issues that might arise.

In providing a more accurate statistical model, Co-Perf should aid in identifying practical implementation issues more accurately. This in turn helps address flaws in current understanding and theoretical software models, which can then be more accurately conveyed to students and practitioners; and finally, data available to consumers can assist in making informed software and hardware purchases.

By making such data available on a wide and cooperative basis, the hope is that Co-Perf will improve the simulation, gaming, and other software industries. Wide availability of such data should provide a unique opportunity to obtain real-world performance data about various system configurations with a higher degree of accuracy.