

Introduction

The PlayStation®2 Performance Analyser (PA) allows developers to see a much more detailed picture of how their programs are running than ever before. In this article I'm going to take a look at some real world examples of how the performance analyser has been used to pick up some obscure performance problems and enabled a developer to really tightly tune their code. I'm also going to point out a few common problems which developers don't notice which the PA highlights well.

Overview

The PA grabs the state of several hundred different debugging signals inside the PS2 every single bus-cycle. This process continually fills a big memory buffer inside the PA, wrapping around every 0.1s or so. The scanning process is stopped at an interesting point, either manually, or using a programmable triggering mechanism. This means that the information left in the memory contains information about what happened in roughly eight or nine frames of a game running at 50Hz. A section of this data is saved out, usually enough for one or two frames of the game loop for analysis – any more is basically a waste of space, but the capacity is useful if a game is badly 'framing out'. The dataset is loaded into a custom application for the actual analysis to take place. The main view of the data is a set of coloured graphs showing the activity of the various signals over a period of time. The whole dataset can be displayed, showing the pattern of activity over the entire capture or the view can be zoomed in to show individual machine cycles – this is necessary to really show the complex interactions between different parts of the PlayStation 2 hardware. There are a large number of graphs showing things such as pipeline activity, VU usage, bus transfers, and the state of the GS pixel units.

Example

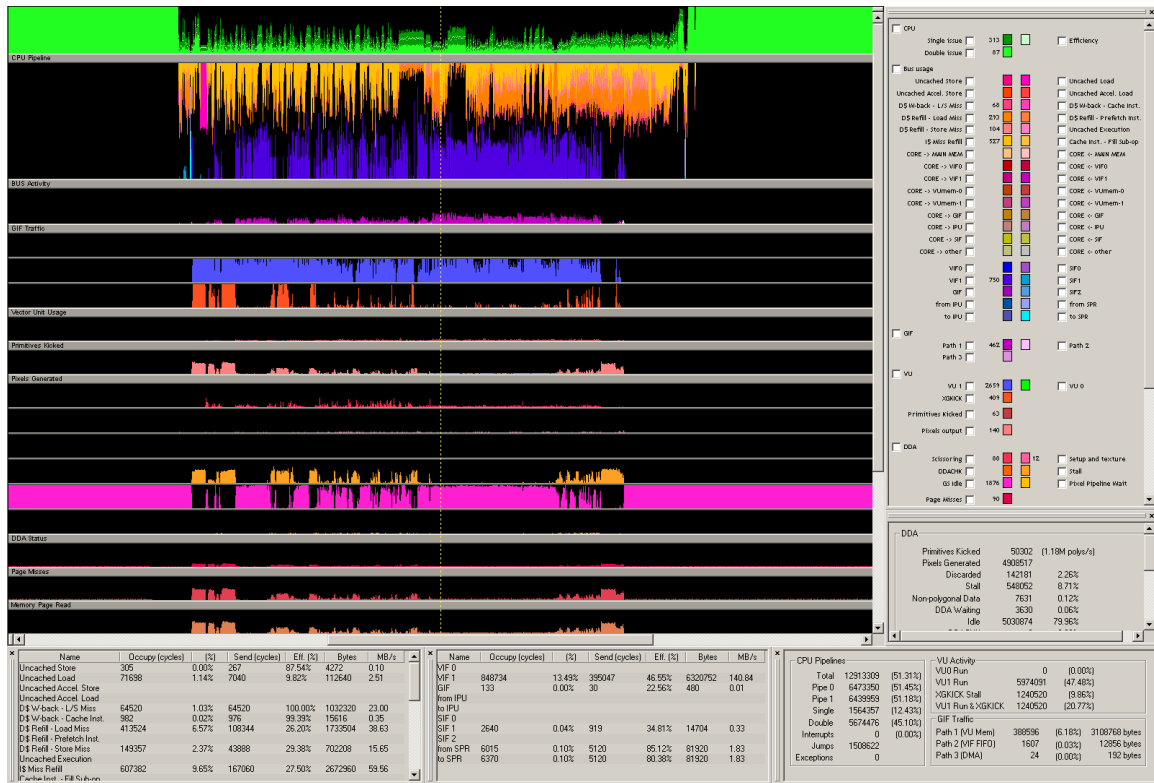


Figure 1 - the typical view seen when the dataset is first loaded into the analyser software.

The scan in figure 1 is of a title achieving relatively average performance, as might be seen before serious optimisation. The main view is surrounded by a whole host of statistics and generally useful information, though this is often more useful for analysing small areas of the scan rather than the whole data set. I'm going to concentrate on looking at the CPU and main bus activity, because that's where the most serious performance problems creep in.

Detail

Looking more closely at figure 1, the very top line (in green) shows the activity of the CPU pipelines. Towards the sides of the region shown this is a solid block – this is a wait loop, which occurs between the processing of each frame. Because this is a busy loop, it looks very efficient, but the graph can be deceptive if you only look at part of the picture. The rest of the graph shows a single frame of the game running, and shows the CPU operating much below the peak level of performance. The black area on this top graph shows the CPU stalling, either because of a hazard condition between instructions (bad scheduling) or waiting on an external resource such as the caches. There are also two different shades of green on this graph. The Emotion Engine is capable of executing two instructions in any given CPU cycle, provided that the next two instructions available to it meet a complex set of criteria. For maximum performance, the compiler (or if coding directly in assembler, the programmer) must be careful to issue instructions which pair well. The lighter green on the graph shows where the CPU was able to pair instructions and the darker green where it could only execute one. When left to its own devices, the compiler generates code which pairs only a fraction of the time, thus the graph shows much more dark green than light green.

The next section of the graph can be used to explain much of the stalling on the CPU pipeline as it shows activity on the main bus. The orange colours represent activity by the CPU, and the blue colours represent various types of DMA access. There is actually a key to the different colours on the right hand side of the screen, but the most important ones to look for here are the dark orange which represents a D-cache load miss, and the lighter orange which represents an I-cache load miss. These are often the two most serious problems with CPU performance other than the scheduling of instructions. Any time data or instructions are required which are not already in the caches, they need to be fetched from main memory, which can be a serious bottleneck. Un-optimised code is unlikely to take access patterns into account, and will rarely get good cache usage. This is quite visible in the scan shown, especially in the last quarter of the frame shown where there is a fairly constant amount of D-cache missing going on. The result is CPU activity which barely rises above 25%.

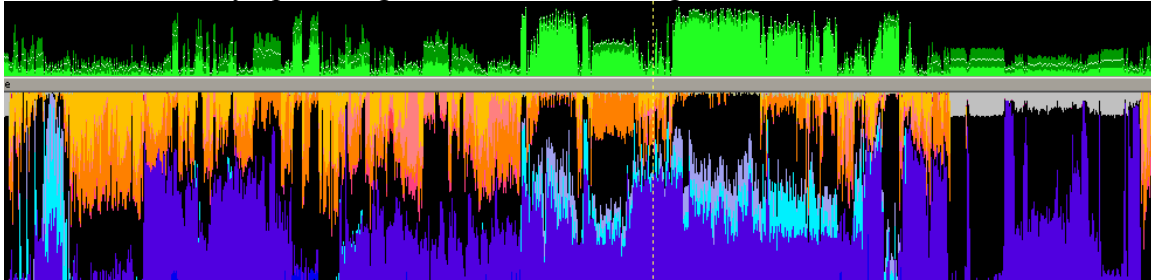
Finding problems

Although DMA activity does not have a direct bearing on the CPU performance, and often is overlooked when CPU optimisation takes place, it is important to realise that all uses of the bus ultimately cause CPU performance issues because it limits the CPU's ability to get data quickly. If the DMA is making heavy demands on the bus, then the CPU often has to wait longer for instruction or data cache refill operations, and as these are already serious stalling points, any delay at all can spell disaster for performance. This is one of the interactions which occur during execution which is tricky to pick up with any kind of intrusive monitoring system. It's also worth considering that it is often easier to optimise the DMA behaviour to make better use of the bus than it is to optimise CPU usage. The result is that it's usually best to optimise DMA usage as much as possible, even when there seems to be more than enough bandwidth available. Although the DMA and rendering speed may not ultimately be affected, the lighter load on the bus will allow the CPU to access random data more effectively.

As you can see, there is a lot of information which can be extracted even from this relatively small part of the data the PA makes available. I simply don't have space in this article to delve into the whole system, but much more detail can be extracted than is covered here.

Optimising

Once all this information is available to a developer, they can really start to do some serious optimisation of their code. The second scan shown here (cropped just to show the CPU and main bus graphs I discussed above) shows the difference in performance which can be achieved by optimising the CPU and bus usage.



Although there is still a fair amount of CPU activity on the bus, especially in the first part of the frame, the DMA bandwidth is reduced enough that it has less of an impact on the CPU performance. Also, significant amounts of scratchpad DMA (in lighter blue) is present, showing the usage of the scratchpad memory to reduce CPU load on the bus – during some very efficient sections the CPU hardly touches the bus at all.

A serious amount of optimisation has clearly occurred for the code executed towards the end of the frame, as the CPU performance is well above average and the instruction pairing is nearly perfect. This is an indication that much of the code has been rewritten by hand in assembly, as no current compiler will generate code this well paired, except in the most simple of loops.

To wrap up, it's well worth using the analyser even if you don't have a serious problem to address. Simply as a tool for understanding the PS2's complex architecture, it's invaluable – but it really comes into its own when trying to squeeze out an extra million polygons before that Easter deadline.