Sony Playstation-2 VPU: A Study on the Feasibility of Utilizing Gaming Vector Hardware for Scientific Computing

By

Pavan Tumati
Advisors: Professor Sanjay Patel and Professor Todd Martinez
ECE298/299

5/16/03

Abstract

One of the driving forces pushing the demand for increased computational power in consumer-oriented devices is the video game industry. Video games, in the past decade, have demonstrated an insatiable desire for computing hardware that accelerates a certain subset of mathematical operations related to efficient rendering of graphical objects on the computer screen. Quite conveniently, these same mathematical operations used to accelerate computer graphics also are capable of accelerating common operations in the realm of scientific computing.

Sony Corporation has invested quite heavily in producing complete computing platforms to accelerate computer graphics. Its offering, the Sony Playstation-2, exposes to software developers a wide array of computational devices that specifically assist in accelerating common graphics related operations, such as vector mathematics. In addition, Sony has offered to the developer community the open-source Linux operating system. The combination of Linux, open source development tools, and detailed developer information allows for the investigation of the feasibility of future use of commodity video gaming hardware for the acceleration of scientific calculations. This document details our investigation and assessment of the potential of the Playstation-2 for scientific computing.

Table of Contents:

Chapter 1 - System Overview, Resources, Expectations and Results

The core of the Playstation-2 is popularly known as the "Emotion Engine." The Emotion Engine is a collection of various subsystems which are, as described in Sony's manual, an effort to "have the highest performance by adopting the latest technology and the most advanced manufacturing technology from the early stages, in order to secure a long product life with performance at the point of sale kept unchanged." Some of the advertised features are fast rendering, multi-path geometry, on-demand data decompression, application specific processors, and data path buffering in what's labeled a "unified memory architecture", or simply UMA.

A block diagram of the overall system substantiates some of Sony's claims:

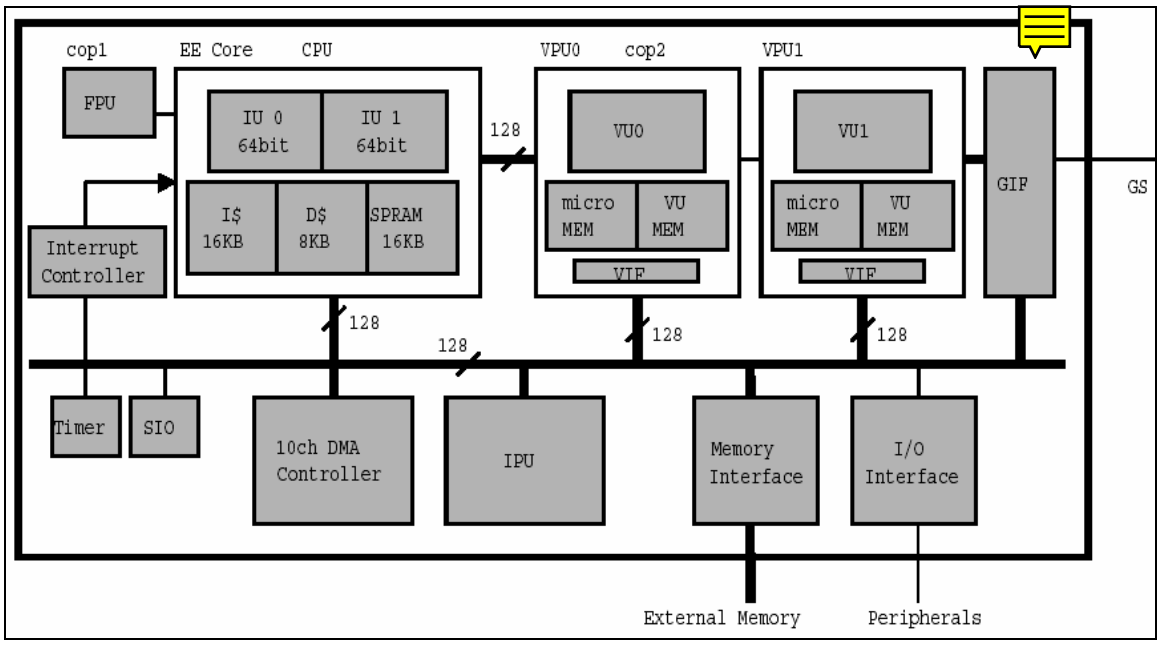

Figure 1: Block Diagram of Emotion Engine Core (Adapted from *EE Overview Manual*, Sony Corporation)

The CPU, Vector Processing Unit (VPU), FPU, scratchpad ram, the instruction and data caches are all collectively labeled the "EE Core."

The CPU implements the superscalar 64-bit MIPS IV instruction set architecture. The instruction cache is 16Kbytes in size, and is two-way set associative. The data cache is 8Kbytes in size, two-way set associative, and supports a write-back protocol.

The EE core features two vector operation processors contained within the VPU Unit for floating point vector operations. These vector units are intended to accelerate geometric calculations. The two vector operation processors within the VPU are known as VU0 and VU1. VU0 is connected to the CPU via a 128-bit wide coprocessor bus. Because of this connection, it is possible to issue to the VPU, from the CPU, what are known as "macro instructions" to do mathematical operations in VU0 hardware. (It is not possible, however, to issue macro instructions to VU1, because of the lack of a connecting bus between the CPU and VU1 registers.) Also, the VU0 and VU1 are connected to separate units, known as vector interface units (VIFs), VIF0 and VIF1. The VIF units can decompress packets of information transferred via DMA into the local memories of VU0 and VU1. It is important to mention that the registers of the vector units are 128-bits wide, and conveniently accommodate vectors containing four 32-

bit floating point elements. Finally, VU1 is connected to the graphics synthesizer (GS) via the graphics interface unit (GIF.)

For scientific computing applications on the Playstation-2, the CPU, VPU, VIF, and DMA units are of primary interest. The primary goal of the research detailed in this document is to utilize the Playstation-2's CPU, VPU, VIF, and DMA units to achieve high throughput for mathematical operations, especially linear algebra types of operations which dominate quantum chemical applications and are expected to be a strength of the PS2 architecture. The most significant computational unit in the PS2 is the VPU; in this document, the CPU, VIF, and DMA are primarily seen as external units whose sole purpose is largely only to keep the VPU "fed" with data. Therefore, it is important to give a brief introduction to the overall architecture and facilities of the VPU.

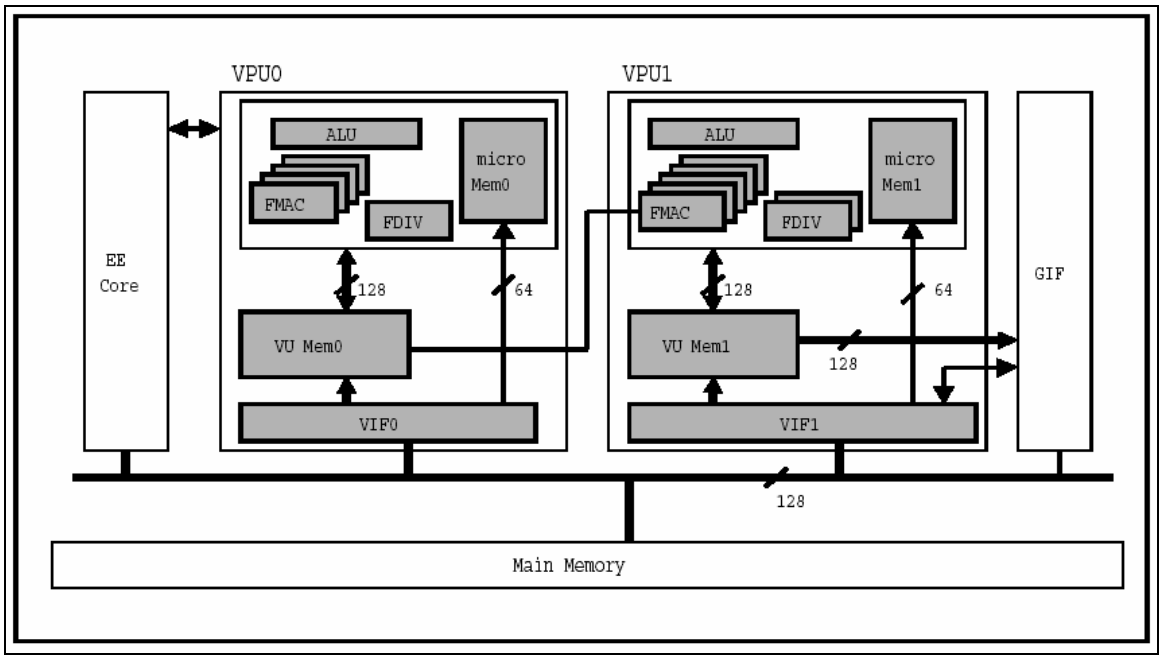

Figure 2: Block Diagram of EE Core Vector Units (Adapted from *EE Overview Manual*, Sony Corporation)

According to Sony's *VU User's Manual*, VU0 and VU1 both have the same basic architecture. That is, both possess data memory (VUmem), a VIF, several floating-point multiply-add ALUs (FMACs), a floating point divide calculator unit (FDIV), 32 four-parallel floating-point registers, 16-integer registers, and program memory (MicroMem.)

There are two general modes of operation for the VPU: macromode and micromode. In macromode, the CPU utilizes coprocessor bus 2 (COP2) to issue "macro instructions" to the VPU, which then utilizes VU0 to complete a given task. In macromode, the CPU sees the registers of VU0 and can transfer information back and forth directly between its own general purpose registers and that of VU0. One of the benefits of having the COP2 connection between the CPU and the VPU is that the CPU does not have to issue a transaction over the main system bus, and can do quick calculations directly.

In micromode, the VPU no longer executes commands issued to it over the coprocessor-2 bus (COP2). Instead, it relies on 64-bit instructions placed in its program memory, or micromem. The 64-bit instructions fed to the VU units are composed of 2 smaller instructions, each 32-bits

in length. The upper 32-bit instruction is fed to the floating-point ALU, and the lower 32-bit instruction is fed to the integer/control unit. The following diagram provides more detail on the structure and instruction issue mechanism of the VUs:

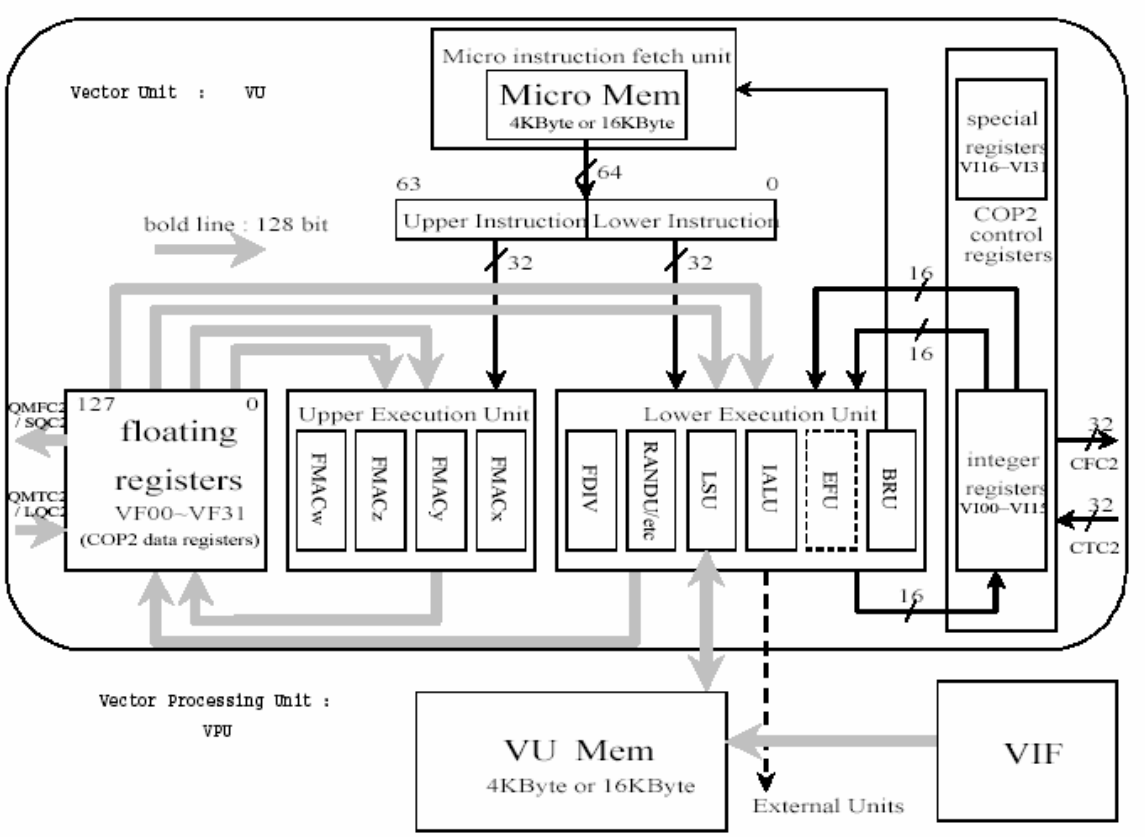

Figure 3: VPU Architecture Block Diagram (Adapted from *VU User's Manual*, Sony Corporation)

Close scrutiny of the diagram above suggests that the primary units of interest are the FMACs, which allow for performing floating-point multiply, addition, and accumulation. It is these units which enable high mathematical throughput for vector operations.

In summary, the PS2s CPU has 2 vector units, in addition to a floating point unit. This is a total of three execution units that can execute in parallel. In addition, the CPU can be utilized in parallel to perform and coordinate computations. Because of this, it is quite evident that there is a great deal of potential for having a high throughput for calculations which fall into the domain of divide-and-conquer, or require several stages of computation.

While the Playstation-2 has already been on the market for quite some time, it is our expectation that systems that are similar to it will be produced in the future and offer even better facilities for increased throughput, with regard to vector-mathematics related calculations. Before proceeding with tests to measure performance, our expectations were to witness a system that would give us reasonable performance that would be on par with the systems that were released around the time of the PlayStation-2. Our results confirm that, in terms of calculation speed, this is indeed the case. However, there were a plethora of performance bottlenecks that complicated *simple* software development. This document details those performance bottlenecks and discusses the feasibility of using gaming systems to accelerate mathematics.

Chapter 2 – Establishing Performance Metric and Basic Test

On examination of the hardware layout of the PS2, it seemed evident that the primary strength of the overall system (as far as its use for scientific computation) was in the VLIW-SIMD vector units provided on the coprocessor-2 bridge to the main R5900-variant processor. The unit, collectively entitled the Vector Processing Unit(VPU), offers vector unit zero (VU0) and vector unit one (VU1). VU0 offers 4K of code and 4k of data memory for VU programs. VU1 offers 16k of code, and 16k of data memory. The instructions processed by the VPU's individual units are VLIW instructions and have 2 components: a floating point unit instruction, and an integer instruction.

The floating-point hardware of each individual vector unit is of key interest; these units feature significant hardware in the form of "Floating Point Multiply and Accumulate Units" (FMACs). It was speculated that scientific applications would benefit the most from the hardware features of the FMAC, and that the overall benefit could be determined through carefully constructed benchmarking software.

Since most scientific code, including computational chemistry, is dominated by floating point math, we aimed to determine the performance benefits of the PS2 for scientific computation using code that relied on floating-point multiplication and accumulation operations. Furthermore, we are especially interested in the performance for numerical linear algebra types of operations, which form the major bottleneck in many quantum chemistry applications. The dot product was chosen as the fundamental operation for our performance tests. This exercises the FMAC units, and further allows straightforward vectorization. Our test is incomplete since it focuses on a single operation, but we point out that the dot product is one of the most difficult linear algebra operations for efficient execution on vector and superscalar architectures because of the large data/computation ratio. Much higher absolute performance, e.g. in terms of MFLOPS, can be expected from matrix-matrix multiplication with its exceptionally low data/computation ratio. This should not be of undue concern in this paper because we emphasize comparisons to efficient code on conventional PC platforms rather than absolute MFLOPS.

All efforts documented in this report focus on utilizing the base dot-product performance test and extending it into different scenarios in which the computational facilities of the PlayStation-2 are explored. It is arguable that this was not the most optimal test; however, considering time constraints, we believed this test would prove useful in assessing basic performance issues involved with the PS2 and would allow us to investigate the hardware in more depth. Essentially, the dot-product performance test was constructed with the idea of determining bounds on FMAC throughput.

As referenced in chapter 1, the VPU can be operated in two modes: macromode and micromode. In order to get a feel for the instruction set of the Sony Playstation-2, it was decided that macromode would be used. In addition, it was decided that macromode code computing dot products could give a general overall picture of a sort of baseline performance that could be expected for the PlayStation-2. The next logical step was to construct or utilize code for computing the dot product and to measure its performance.

Chapter 3 – Macromode Performance Analysis and Test Construction

In macromode, the principal components exercised within the Emotion Engine are the CPU, the COP2 bus, and VU0. In the diagram below, the exercised units are highlighted:

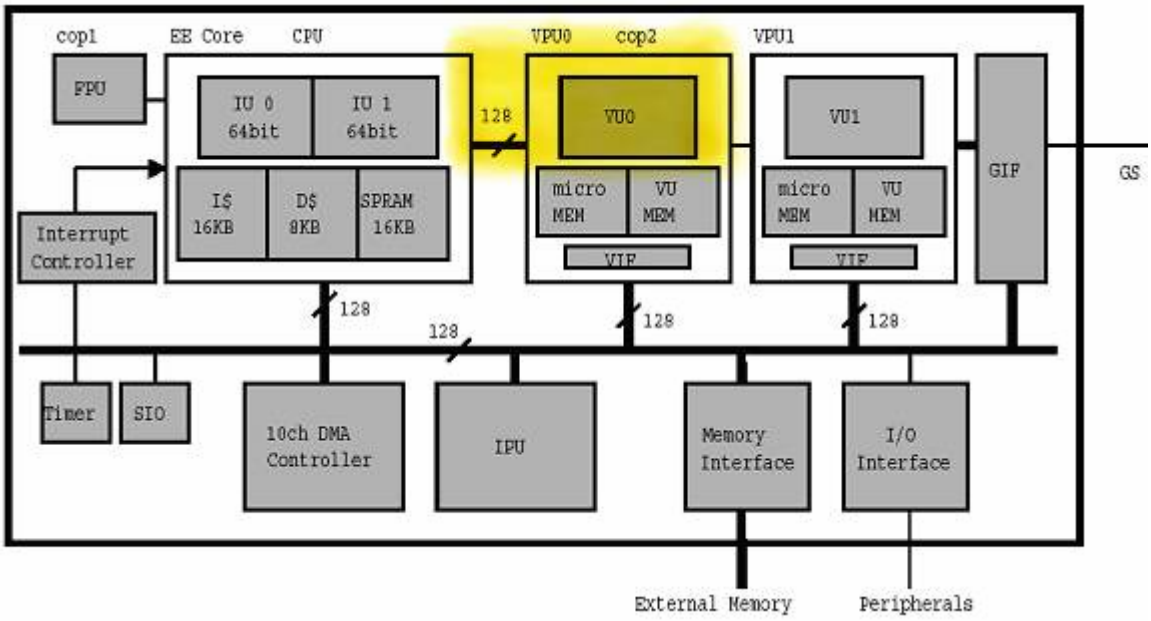

Figure 4: EE Core COP2 and VU0 Location (Adapted from *EE Overview Manual*, Sony Corporation)

In order to understand and exercise the VU0 unit, a small, crafty piece of code needed to be constructed or found. In this chapter, the objective of the code below, obtained from *http://www.bd.wakwak.com/~yumimint/ps2linux/vu-tips.html* , is to come up with a scalar result of the dot-product of two four-element vectors.
For the purposes of demonstrating an example, supposing there were two vectors v1 and v2:

v1 = < x1, y1, z1, w1 >
v2 = < x2, y2, z2, w2 >

To compute the dot product, code would be written that is functionally similar to what is listed below. (Note that the registers prefixed with "VF" to their name are macromode references to the floating point registers in VU0, the unit that gets exercised when requests are made over coprocessor bus 2.)

```
float COP2_DotProduct( void *src1, void *src2 )
{
  float result;
  __asm__ __volatile__
 ("
   lqc2         vf16, 0x0( %1 )
   lqc2         vf17, 0x0( %2 )
   vaddw.x      vf18, vf00, vf00
   vmul.xyzw    vf16, vf16, vf17
   vmulax.x      ACC, vf18, vf16x
   vmadday.x     ACC, vf18, vf16y
```

```
   vmaddaw.x    ACC, vf18, vf16w
   vmaddz.x    vf16, vf18, vf16z
   .set noat
   qmfc2      $1, vf16
   mtc1       $1, %0
   .set at
 "
 : "=f" (result)
 : "r"(src1), "r"(src2)
 : "$1"
 );
  return result;
}
```

The code above doesn't make much sense until the notation for the various macro instructions is understood. Before reading the analysis below, it is important to note a few things. In the VPU of the PS2, the floating point registers have 4 32-bit parts. They are termed the w, z, y, and x components.

- - The w component refers to bits 127-96.
- - The z component refers to bits 95-64
- - The y component refers to bits 63-32
- - The x component refers to bits 31-0

Table 1: Vector Component Format

W (127-96)	z (95-64)	y (63-32)	x (31-0)

The VF00 register, however, has its components set to fixed values. The w value is set to 1.0, and all of the other parts of the register are set to 0.0.

For the purpose of the understanding the detailed analysis below, assume that the contents of VF16 correspond to the parts of a vector v1 <x1, y1, z1, w1> and the contents of VF17 correspond to the parts of a vector v2 <x2, y2, z1, w2>. The "dot" after a register name in conjunction with a component letter is a reference to an element of a vector. For example, VF16.x refers to the x component of the 4 floating-point element vector contained within register VF16.

The analysis of the code above is as follows:

Table 2: VU Macromode Dot Product Computation Code Breakdown

Instruction	**Algebraic Expression**	**Important Things to Note**
`lqc2 vf16, 0x0( %1 )`	`VF16 = 4 floating point values from parameter src1`	
`lqc2 vf17, 0x0( %2 )`	`VF17 = 4 floating point values from src2`	
`Vaddw.x vf18, vf00, vf00`	`VF18.x = VF00.x + VF00.w`	`VF18.x = 1 (Note VF00.x=0, VF00.w = 1, always )`

`Vmul.xyzw vf16, vf16, vf17`	`VF16.x = VF16.x * VF17.x`
	`VF16.y = VF16.y * VF17.y`
	`VF16.z = VF16.z * VF17.z`
		`VF16.x = x1*x2`
		`VF16.y = y1*y2`
		`VF16.z = z1*z2`

	VF16.w = VF16.w * VF17.w	VF16.w = w1*w2
Vmulax.x ACC, vf18, vf16x	ACC.x = VF18.x * VF16.x	ACC.x = x1 * x2
Vmadday.x ACC, vf18, vf16y	ACC.x = ACC.x + VF18.x * VF16.y	ACC.x = x1 * x2 + y1 * y2
Vmaddaw.x ACC, vf18, vf16w	ACC.x = ACC.x + VF18.x * VF16.w	ACC.x = x1 * x2 + y1 * y2 + w1 * w2
Vmaddz.x vf16, vf18, vf16z	VF16.x = ACC.x + VF18.x * VF16.z	VF16.x = x1 * x2 + y1 *y2 + w1 * w2 + z1 * z2
Qmfc2 $1, vf16	$1 = VF16 (VU to EE data transfer)	$1 = VF16 Contents
Mtc1 $1, %0	Store lower 32 bits of GPR into floating point register. (We constrained %0 to a floating point register via "=f" specification in gcc's extended asm syntax.)	Transfer the floating point data to the value we want to return.

It's quite clear from the column showing the algebraic expression that the dot-product operation benefits immensely from the SIMD characteristic that allows for 4 single-precision floating point multiplications in 1 instruction. The gcc-inlined code above featuring accumulation operations, result retrieval (qmfc) code, and the actual dot-product computation code allow for us to then construct a test to measure the actual performance of the code in execution.

In order to compute the effectiveness of the above macromode code and to compare overall throughput with that of other commodity systems, the function above was placed in "competition" with similarly functioning C-code compiled for other platforms. Notice that the assembly code above is encapsulated in a C-function, "COP2_DotProduct()." In non-PS2 systems, this function was replaced with a more generic C function used to compute the dot-product. The competition code then was run repeatedly to figure out how fast a given system could compute dot-products. All test programs were compiled using gcc for the respective architecture.

It is important to note that in the utilized performance program, on non-PS2 test systems, the equivalent dot product calculation code did not utilize specialized SIMD instructions, such as IA32 MMX/SSE/SSE2 or G4 AltiVec instructions. It was decided that the performance tests would pit the PS2 facilities against "average" or generically compiled mathematical C programs. The general idea was to compare the PS2 code with code that was generically compiled by a C program on the IA32 and G4 architectures; the rationale behind this decision was that there would be specific advances in compilers and toolkits for providing a better, overall "generic" performance on the Playstation-2 for average mathematical programs relying on single-precision floating point arithmetic.

The performance graph below (obtained through the use of gprof) was obtained. (Note that the x-axis represents the number of dot products performed, and the y-axis represents the time required to complete all the calculations.)

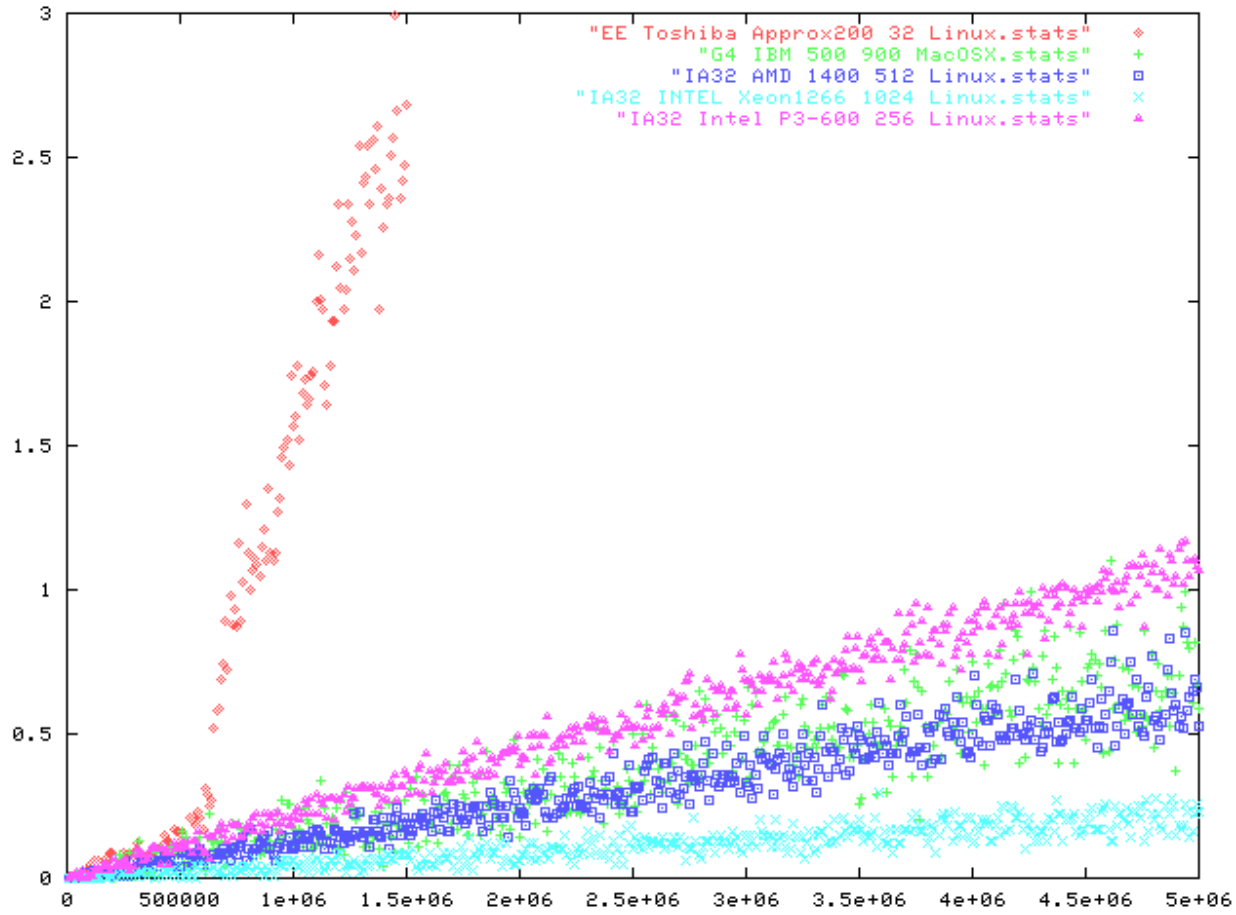

Figure 5: Playstation-2 Dot-Product Calculation Performace versus Macs and PCs (X axis = # of Dot Products, Y axis = time in seconds.)

It is noticeable that a generally linear trend in performance with the increase in the number of dot products being calculated. Also quite evident is the sharp performance drop off of the PlayStation-2 after a certain data-size threshold is reached. This performance drop off is considerable, and demonstrates that the 32MB integrated RAM will prove to be a significant bottleneck because of paging. Consequently, it can be stated: *any utilization of the PlayStation-2 for efficient mathematics must avoid paging penalties and disk I/O.* Notice, however, that in the region where the PS2 was not paging, it substantially outperformed other processors in terms of OPs/cycle. Therefore, tests designed to measure the raw computational throughput of the PS2 hardware were refactored to avoid exercising the scenario in which the virtual memory and disk subsystem were exercised.

Chapter 4 – Micromode Performance Analysis

Looking at the performance graph for macromode, it is evident that macromode performance is fairly decent in comparison to commodity hardware released at around the same time the PS2 was released (given the test conditions stated above with regard to optimizations), but does not allow us to take full advantage of the advertised hardware offerings of the PlayStation-2. It was speculated during experimentation that utilization of micromode would allow for a significant performance boost that would allow our performance graphs to reach the performance levels of more common (not necessarily top of the line) PC hardware.

The primary difference in micromode code utilization is that the CPU no longer specifically issues instructions over the coprocessor bus; instead, the VUs operate completely independently and in parallel. In addition, in macromode, only 1 VU's FMACs are used. By switching to micromode, we gain the ability to exercise the FMACs of both VUs. Because of the utilization of 2 VUs, it was expected that the performance throughput would roughly double. This idealized expectation neglects any contributions from the CPU or the FPU. The diagram below shows the portions of the PS2 that were used in the VU performance tests:

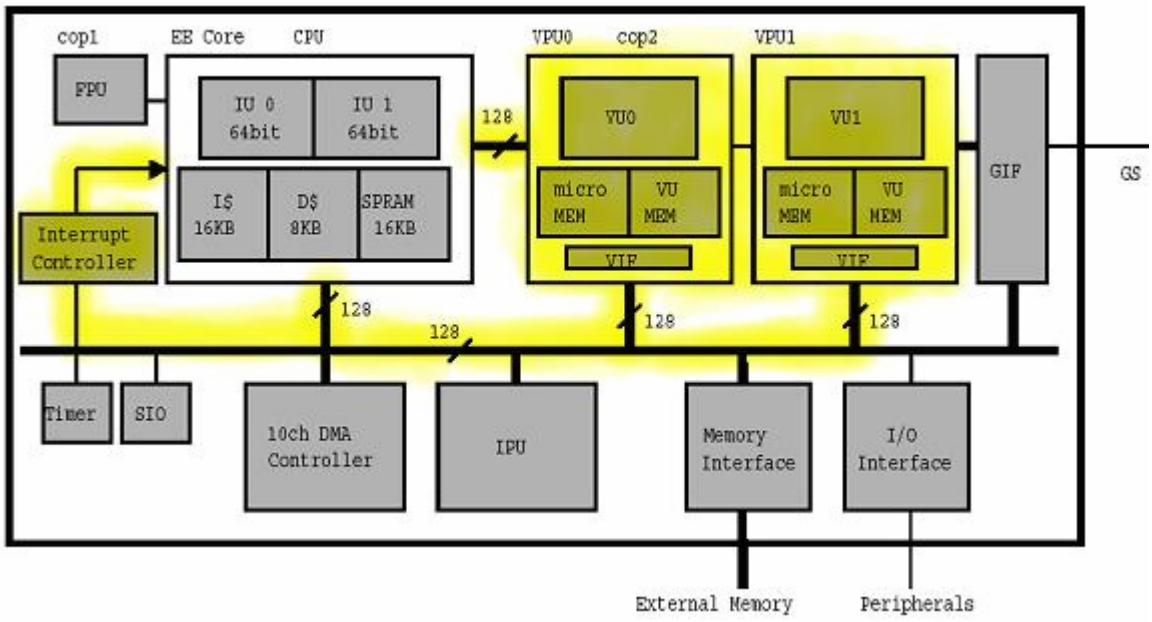

Figure 6: EECore Components Used in Micromode Tests (Adapted from *EE Overview Manual*, Sony Corporation)

The primary downside, initially, of micromode performance analysis was that entirely new program code needed to be written and assembled so that data could be written into the VUs. This meant that control instructions would have to be separated from mathematical instructions, and that completely new code would have to be written and assembled using a separate assembler. No longer could instructions be issued to the VUs directly; instead, the CPU needed to take pre-assembled VU instructions and write those instructions into the micromem of each individual VU unit.

Because the operating system utilized was Linux, software executing in unprivileged user-mode needed to execute a mmap() system call on Sony-issued device drivers that provided a convenient interface to user-mode applications. The mmap() function invoked on the /dev/ps2vpu0 and /dev/ps2vpu1 devices effectively adjusted the page tables associated with a process' virtual address space to enable non-cached page-mappings corresponding to the physical address space memory corresponding to the VUmem and micromem of the individual VUs. Our test process had an address space, described by the 'maps entry' in the /proc filesystem, somewhat similar to the following:

```
00300000-00301000 r-xs 00000000 03:02 99703      /dev/tst
00400000-00409000 r-xp 00000000 03:02 4636971    ~/code/performance/performance.micromode.comb
0fb60000-0fb79000 r-xp 00000000 03:02 229378     /lib/ld-2.2.2.so
0fb88000-0fb89000 rw-p 00018000 03:02 229378     /lib/ld-2.2.2.so
0fb89000-0fb8a000 rwxp 00000000 00:00 0
10000000-10001000 rw-p 00009000 03:02 4636971  ~/code/performance/performance.micromode.comb
10001000-1000b000 rwxp 00000000 00:00 0
2aaab000-2aaac000 rw-p 00000000 00:00 0
2aaac000-2aaae000 rw-s 00000000 03:02 99073      /dev/ps2vpu0
2aaaf000-2aaf3000 r-xp 00000000 03:02 229389     /lib/libm-2.2.2.so
2aaf3000-2ab32000 ---p 00044000 03:02 229389     /lib/libm-2.2.2.so
2ab32000-2ab34000 rw-p 00043000 03:02 229389     /lib/libm-2.2.2.so
2ab34000-2ab46000 r-xp 00000000 03:02 229407     /lib/libpthread-0.9.so
2ab46000-2ab85000 ---p 00012000 03:02 229407     /lib/libpthread-0.9.so
2ab85000-2ab8c000 rw-p 00011000 03:02 229407     /lib/libpthread-0.9.so
2ab8c000-2ab8d000 rw-p 00000000 00:00 0
2ab8d000-2acbf000 r-xp 00000000 03:02 229383     /lib/libc-2.2.2.so
2acbf000-2acfe000 ---p 00132000 03:02 229383     /lib/libc-2.2.2.so
2acfe000-2ad07000 rw-p 00131000 03:02 229383     /lib/libc-2.2.2.so
2ad07000-2ad0c000 rw-p 00000000 00:00 0
2ad0c000-2ad14000 rw-s 00000000 03:02 99074      /dev/ps2vpu1
2ad14000-2b763000 rw-p 00000000 00:00 0
2b763000-2b764000 rw-p 00002000 03:02 4636907    ~/code/performance/dp.elf
2b764000-2b765000 rw-p 00000000 03:02 99068      /dev/ps2mem
2b765000-2b766000 rw-p 00002000 03:02 4636907    ~/code/performance/dp.elf
7ffee000-80000000 rwxp fffef000 00:00 0
```

As mentioned in chapter 1, the VUs expect instructions 64-bits at a time. Each instruction is actually a pair of 32-bit instructions and these are fed individually to the floating point unit and the integer unit. Thus, the macromode code described above needed to be rewritten appropriately. The following code, logically similar to the macromode code, was constructed. In an attempt to optimize the code, the loop was unrolled manually such that 4 dot products were computed in one loop iteration. Notice also that branch-delay slots are supported in the VUs and are utilized in the following code:

```
.vu
.data

# argument 1: <size of dot product chunks in terms of vector count >
# argument 2: <location of data chunk 1>
# argument 3: <location of data chunk 2>
# argument 4: 0 -> Status Location

          .text
          .globl uModeDotProduct

uModeDotProduct:

# First, the parameters need to be extracted:

# vi1 contains the number of dot products we are calculating
# vi2 contains the location of dot product array 1
# vi3 contains the location of dot product array 2
# vi4 contains the destination address of the results
```

```
# Low order address -> w, high is x

  sub.xyzw vf1, vf00, vf00              ilwr.x     vi1, (vi0)x
  nop                                   ilwr.y     vi2, (vi0)y
  nop                                   ilwr.z     vi3, (vi0)z
  nop                                   ilwr.w     vi4, (vi0)w

##### Stage 1, Part of Stage 2 P1[16, 17, 18] -> P2[ 19, 20 ]

# First attempt/pass adaptation of macromode computation code
# for micromode code.
  nop                                   lqi.xyzw vf16, (vi2++)
  nop                                   lqi.xyzw vf17, (vi3++)
OuterLoop1:
  addw.x     vf18, vf00, vf00           iadd      vi5, vi4, vi0
  mul.xyzw   vf16, vf16, vf17           sqi.xyzw  vf1, (vi5++)
  mulax.x     ACC, vf18, vf16x          lqi.xyzw vf19, (vi2++)
  madday.x    ACC, vf18, vf16y          lqi.xyzw vf20, (vi3++)
  maddaw.x    ACC, vf18, vf16w          nop
  maddz.x    vf16, vf18, vf16z          nop
  addw.x     vf21, vf00, vf00           mr32.w   vf15, vf16
  mul.xyzw   vf19, vf19, vf20           mr32.z   vf15, vf15
  mulax.x     ACC, vf21, vf19x          lqi.xyzw vf22, (vi2++)
  madday.x    ACC, vf21, vf19y          lqi.xyzw vf23, (vi3++)
  maddaw.x    ACC, vf21, vf19w          lqi.xyzw vf25, (vi2++)
  maddz.x    vf19, vf21, vf19z          lqi.xyzw vf26, (vi3++)
  addw.x     vf24, vf00, vf00           mr32.w   vf15, vf19
  mul.xyzw   vf22, vf22, vf23           lqi.xyzw vf16, (vi2++)
  mulax.x     ACC, vf24, vf22x          lqi.xyzw vf17, (vi3++)
  madday.x    ACC, vf24, vf22y          mr32.yz  vf15, vf15
  maddaw.x    ACC, vf24, vf22w          nop
  maddz.x    vf22, vf24, vf22z          nop
  addw.x     vf27, vf00, vf00                  mr32.w   vf15, vf22
  mul.xyzw   vf25, vf25, vf26           mr32.xyz vf15, vf15
  mulax.x     ACC, vf27, vf25x          nop
  madday.x    ACC, vf27, vf25y          nop
  maddaw.x    ACC, vf27, vf25w          nop
  maddz.x    vf25, vf27, vf25z          nop
  nop                                   mr32.w   vf15, vf25
# Terminate execution and transfer control back to VU
  nop                                   isubiu   vi1, vi1, 4
  nop                                   ibgtz    vi1, OuterLoop1
  nop                                   sqi.xyzw vf15, (vi4++)

  nop[t]                                nop
  nop[e]                                nop
  nop                                   nop
  nop                                   nop
```

For the sake of brevity, a more detailed instruction by instruction analysis is not provided here, but is provided in Appendix A. However, it can be witnessed that the same general computation code structure from the macromode example is presented in the left-hand column. The left-most instruction stream is, in fact, the stream of instructions that are issued to the floating point hardware. The right-most stream of instructions is issued to the integer/control unit. Notice that at the end of the instruction stream, there are No-Operation instructions (nops) with ‘t’ and ‘e’ flags specified. These instructions instruct the VU to raise an interrupt to signal to the CPU that computations have been completed. (The software used to conduct performance tests needed to wait on a queue until a secondary interrupt service routine awakened them.)

In the initial test of the above code, only VU0 was used to measure performance. Eventually tests were performed utilizing VU1 alone, and also both VU0 and VU1. The following table relates performance of macromode with micromode in terms of time required to perform 200,000 dot products:

Table 3: VU Performance In Various Modes In Terms of Time for Calculating 200,000 dot products

Mode	Units used	Exec Time
Macro	VU0 (via COP2)	.144 seconds
Micro	VU0	14.8 seconds
Micro	VU1	3.1 seconds
Micro	VU0+VU1*	3.5 seconds

**Data was transferred to the VUs only after both VUs completed processing their data.*

Macromode outperforms micromode! The significant performance difference is attributable to the amount of time spent transferring data across the main memory bus from RAM to the data memories of the VU units. (Note: It is important to note that the tests within this chapter relied on the CPU to perform data-transfers via the LQ and SQ opcodes, and did not rely on DMA and VIF hardware.) In addition, the number of system calls required for the user-mode code to control VU units contributed to an increased overhead.

The amount of time required to transfer from user-mode to kernel-mode for system calls to VPU device drivers is significant. Latencies in the kernel control paths also contributed to performance degradation. Clearly, in order to avoid the performance penalties of data transfer and system-call overhead, another solution was required. Note, also, that the dot-product performance test lends itself to this behavior because the VU code does not systematically-reuse the data passed to it. It is speculated that if the ratio of transfer time to computation time were adjusted accordingly, significant performance increases would be realized.

To get a feel for what sort of performance would be obtained *without* the transfer penalty, but *with* control-overhead a test was devised in which the VUs individually computed dot products over and over, without transferring data back and forth to RAM. The resulting time for 200000 dot products is:

Table 4: VU Micromode Performance Without Data Transfer Overhead

Mode	Units used	Exec Time
Micro	VU0+VU1*	3.0 seconds

For increased performance, significant modifications would be required to avoid kernel-control path and system call latencies. The assisting in proving the hypothesis that system-call overhead was the primary bottleneck, the code was restructured in a manner that allowed for calculation of 200,000 dot-products without any data transfers or raised-interrupts. In other words, to create the ideal situation without control-overhead and data-transfer overhead, the VU code was rewritten so the VU would repeat over 1 buffer and compute the dot-product repeatedly 200,000 times. This resulted in 200% improvement over macromode code, using only VU1! The results are shown in Table 5.

Table 5: VU Performance for 200,000 Dot Products without Control and Data Transfer Overhead

Mode	Units used	Exec Time
Micro	VU1	.07 seconds.

Chapter 5 – Micromode VPU + FPU Usage Analysis

When the performance test software for determining the values in the previous section was designed, it was speculated that perhaps some benefit would arise from exploiting multiple threads of execution. The idea, simply, was to attach a control-thread, instantiated via the Linux p-threads interface, to each computational device. For example, in the previous section, if both VUs were utilized at the same time, VU0 always finished before VU1, and would sit idle until the software was finished. One thread would monitor and feed the information to VU0, one would monitor VU1, and yet another would be used to feed the FPU connected to the main processor.

In the last chapter it was quite evident that the control-overhead was a primary bottleneck in achieving high performance out of the VPU. Therefore, it suffices to say that the added overhead of multiple threads did not alleviate this bottleneck, but instead exacerbated the problem – at least, with regards to the dot-product test. However, it is suspected that, for software that requires a long compute time with respect to control-overhead time, a threaded software model would be beneficial in many cases, especially when dividing-and-conquering mathematical problems.

The MIPSLinux kernel provided with the Sony Playstation-2 kit does not support utilizing the VPU with multiple-threads. That is, during user-mode process context switches, the registers of the VPU are not saved. Furthermore, only one user-mode thread of execution may have ownership of the VPU at any given time. To alleviate this problem and to perform tests on multi-threaded applications, an investigation was performed on the management of the FPU inside the kernel and a solution for the VPU was determined.

In most user-mode processes in Linux, the kernel does not bother to save the floating point registers on a context switch. This is a performance enhancement in that the amount of time required for a context switch in user-mode decreases if the floating point registers do not have to be saved. For processes that require floating point instructions, what happens is that the user-mode process' first FPU instruction generates an exception. The exception handler then appropriately sets the bits required in process-specific data structures in the kernel and enables access to the FPU. After the process-specific data structures are modified, the kernel then saves the floating point registers on a context switch.

For adding multi-threaded support for the performance test software and other applications, two approaches could have been utilized: 1) Manage the VPU like the FPU and save the VPU registers visible over the COP2 bus on a context switch, or 2) Designate VPU-control code as critical sections, and guard them with mutual exclusion constructs. Option 1 was the more complex solution, and was avoided in the short term. Option 2 was studied in this document.

In order to allow for multiple-threads to utilize the vector units at a time, a device driver (vpuperm) was written to facilitate access to the VUs. In Linux, whenever a user-mode to kernel-mode transition happens via the system call mechanism, the kernel saves all of the registers of the process to the stack. The vpuperm device driver simply modifies the permissions of threads which possess active vpuperm-file-descriptors by adjusting values on the stack. The system-call utilized was ioctl(), and the basic code is presented here:

```
void EnableCOP2( int stat, unsigned char *frameptr )
{
```

```
  KFEND( PRIORITY1, "EnableCOP2()" );
#ifdef __MIPSCOMPILATION__
  _KDbgPrint( ( "VPUPERM: Previous Status: %08X\n",
                *((unsigned int *)( frameptr + PT_STATUS ) ) ) );

  if( stat )
    *((unsigned int *)( frameptr + PT_STATUS )) |= ST0_CU2;
  else
    *((unsigned int *)( frameptr + PT_STATUS )) &= ~ST0_CU2;

  _KDbgPrint( ( "VPUPERM: Current Status: %08X\n",
                *((unsigned int *)( frameptr + PT_STATUS ) ) ) );
#endif

  KFEXD( PRIORITY1, "EnableCOP2()\n" );
}
```

Again, it should be mentioned that the control overhead for the dot-product test was so significantly high, that multi-threaded test code results in severely degraded performance. The key benefit from threads in the PS2 arises when VU computational time far exceeds the cost (in terms of time) to perform control operations and transfer data. In the future, it is expected that more tests can be constructed to utilize multiple threads.

Chapter 6 – Alleviating Bottlenecks in Performance

It's painfully evident from the results in chapter 4 that eliminating system call overhead is critical for VU utilization. The delays witnessed in existing performance tests are primarily due to the structure of the performance testing software and the device driver utilized.

The designed dot product performance test software currently executes in user-mode, and relies on a set of ioctl()'s to query the VU hardware. The performance test software typically starts the VU transaction directly over the COP2 bus, but sleeps on a wait-queue in the kernel until the VU executes the nop[t] instruction. The overhead associated with the user-mode to kernel-mode transition, in conjunction the event-management and thread-wake-up code is a hefty price to pay in the dot-product test.

Consider VU0 -- it has 4k of data memory. Each vector requires 16-bytes of space. The data memory needs to be utilized to contain the two vectors used for the dot-product, the scalar results, and parameters passed into the VU functions. Thus, the VU0, in the performance test used, was only able to compute 36 dot-products before requiring an interrupt. The VU1 unit performs only 4x as many computations before requiring an interrupt to notify the CPU that more data needs to be sent across the bus. The quick computation time of the VUs in relation to the overall overhead incurred from managing interrupt service routines and user-to-kernel transitions causes the overall process to be inefficient.

A simple potential solution would be to remove the system-call overhead within the kernel, and to simplify event management code within the PlayStation-2 Linux kernel device driver. Alternatively, the software developer can elect *not* to use the VUs when the ratio of system-calls and interrupts to VU computation time is unfavorable. In addition, the developer can simply expose all of the registers of the VU hardware to user-mode code via mmap(), and eliminate system-call overhead in order to manage everything in user-mode. (This strategy is used by the popular open-source *SPS2* library.) The disadvantage of this is…

Before investigation into initiatives requiring the restructuring of device drivers and kernel management code, a few other alternatives were investigated on the Playstation-2. Namely, the DMA and VIF interface was inspected. At this time, no test has been developed to show an increase in performance, but because of the results of the previous chapter, it is speculated that DMA and VIF would play a vital role in increasing overall PS2 throughput.

On the PlayStation-2, it is possible to construct a list of information containing a sequence of floating points values or integers that can be transferred into the code and data memories of the VUs. These lists consist of what are called "VIF-packets." VIF-packets are constructed in such a manner that they can be used with the DMA hardware. That is, arbitrary data and commands can be streamed to the vector units. Interestingly, VU0 can only have data DMA transferred to it, but does not support DMA transfers from its data memory back to the CPU. VU1 and VIF1 do support transfers in both directions.

In order to communicate with the vector units through the VIF hardware, 128-bit aligned VIF packets need to be constructed. Within the packet, a control operation is specified through what is known as a VIF-code. There are several types of VIF-codes that are embedded in VIF packets. For example, codes can be a NOP, a data transfer with the UNPACK command, a

microprogram transfer to code via the MPG command, or even a function call of sorts to force the VU to start executing code from a specific address in the code memories.

Within the dot-product performance test, the dominant VIF operation was the UNPACK operation, which places information into the data memories of the VU. The dot-product performance test placed the microprogram into the code-memories in advanced, so no MPG operation ever needed to be issued.

The VIFCode UNPACK packet is structured as follows (in C struct description):

```
typedef struct
{
  unsigned int addr     : 10;  /* Bits   0-9, Start Address                */
  unsigned int reserved :  3;  /* Bits 11-13, Reserved                     */
  unsigned int usn      :  1;  /* Bits    14, Signed Unsigned Decompression */
  unsigned int flg      :  1;  /* Bits    15, Address Mode - Add VIF1_Tops? */
  unsigned int numcount :  8;  /* Bits 16-23, Number of 128 bit chunks      */
  unsigned int cmdcode  :  8;  /* Bits 24-21, Command Code                 */
}VIFCodeStruct_unpack;
```

The VIFCodeStruct_unpack information needs to be placed ahead of the actual data in the buffer being transferred. The number of 128-bit chunks being transferred is specified within the packet information. By toggling bits in the actual command code field, the VIF unit can be instructed to raise an interrupt to signal the end of a data transfer. In the dot-product test performance application, this interrupt was used to signal that more information needed to be transmitted. Unfortunately, in current versions of the performance test, the utilization of the interrupt caused increased performance overhead, so no real performance benefit was gained in existing performance tests. The generalized code for performing a data transfer using Sony provided VPU device drivers and the UNPACK command is presented:

```
  unsigned int vifstreamheader[] =
    { 0x00000000, 0x00000000, 0x00000000, 0x6C000000 };

[…]

  dmapacket.ptr = (void *) fpacket;
  dmapacket.len = sizeof( VIF0FullTransferPacket );

  vifevent = VIF0EventOpen();
  if( vifevent < 0 )
  {
    printf( "Unable to open VIF event object.\n" );
    __RETURN( vifevent, "main()" );
  }
  memset( (void *)( vu[0].datamem ), 0, PAGESIZE );
  ioctl( vu[0].fildes, PS2IOC_SENDA, &dmapacket );
  printf( "Transfer complete.\n" );
  VIF0EventWait( vifevent );
  VIF0EventClose( vifevent );
```

In the above code, the ‘fpacket’ variable is a pointer to a block of floating point data containing a copy of the ‘vifstreamheader’ array (with the VIF UNPACK code encoded in it) and a block of numbers to be used in a dot-product calculation. When the ioctl() is performed, the non-blocking PS2IOC_SENDA command (enumerated in header files provided with the source code for the Sony PS2-Linux kernel VPU device driver) is specified. This command instructs the device driver to perform a DMA transfer of length specified in dmapacket.len, from the address specified in dmapacket.ptr. The dmapacket.ptr in this case points to the ‘fpacket’ region which

contains proper information formatted for the VIF, so that information can be placed in the VUs data memory.

Notice that for the transfer to complete successfully, the ioctl() was required to start the transfer and computation, and that another system call was required to wait for the transfer to complete. The VIF0EventWait() code is just a wrapper function for a blocking system call (via ioctl()) that forces the thread to sleep until the data transfer is complete. The current test requires two system calls to coordinate operation. Again, the existing control overhead is high. However, it is speculated if the two system calls were removed, and the control were localized more effectively to strictly kernel space, the VIF transfer method would prove optimal and also allow for the CPU to be re-utilized during DMA transfers for performing other operations such as more dot-product calculations. This is the ideal scenario, and is the objective for the development of a new testing and performance measurement suite.

It is speculated that if the user-mode to kernel-mode system-call overhead were eliminated, and that the interrupt-management overhead that currently exists is eliminated, that the VIF transfer mechanism would be exceptionally efficient. Ideally, in a restructuring of performance test code, the Sony device drivers would be rewritten in such a manner that the kernel would have copies of the buffers used for computing dot-products within kernelspace, and the interrupt management could would be more streamlined, such that no extra computation would have to be performance to determine which user-mode process thread would need re-awakening upon completion of VPU mathematical processing after a DMA transfer.

In gaming software, the level of organization and sophistication of device drivers is miniscule; however, the organization required to manage information within the Playstation-2 Linux kernel is much greater. In order to successfully utilize Linux, the time-to-hardware for mathematical data must be reduced, and certain protections offered by a Unix-like OS will need to be traded off.

Chapter 7 – Possible New Directions in Performance Analysis and Test Development

In the previous chapter, direct transfer of data to the VU data memories via the VIF units was discussed. It's evident that, even after data transfer and execution of dot-product code, information must still be reclaimed via memory copy cycles, or another DMA transfer. The user-mode code still has quite a bit of system-call overhead to initiate the DMA transfers. Clearly, to remove system-call overhead, more functionality needs to be integrated into the device drivers for the vector units, or computational code needs to be placed within kernel-space.

With regards to the dot-product performance test, the ratio of control-overhead to computation time is still too high. Another alternative to the techniques previously used is to adjust the flow of data such that, instead of reclaiming the results from the vector units after computation, the data be transferred as "texture" data to the graphics synthesizer and its local memory via the GIF unit. The Playstation-2 graphics unit has 4MB of local memory in which textures can be stored, and through which data can be transferred back and forth via DMA and through "KICK" instructions through the vector units. The new region of the EE core that would be utilized is highlighted, in addition to other units used in micromode, in the block diagram below:

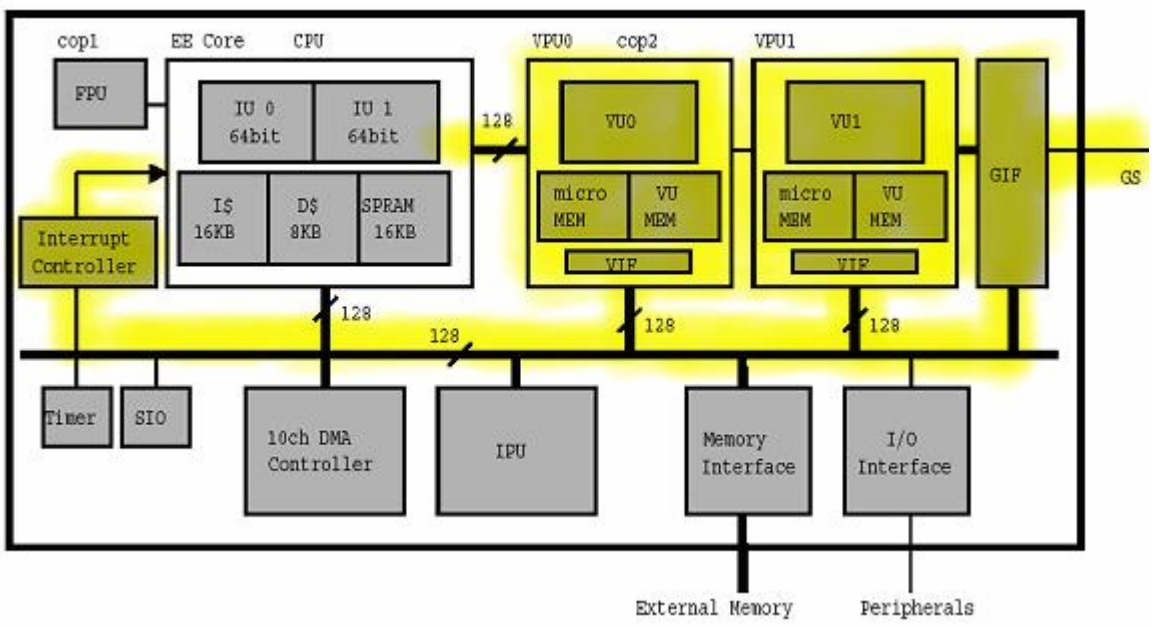

Figure 7: GIF and GS Unit location (Adapted from *EE Overview Manual*, Sony Corporation)

The graphics synthesizer introduces a great deal of complexity: textures must be managed, the local memory must be partitioned, and the hardware must be instructed not to perform any transformations on the incoming data. Future tests are expected to utilize the graphics synthesizer in the overall performance test. The development time for such a test is anticipated to be very high, due to the high configurability of the graphics synthesizer.

Chapter 8 – Results and Feasibility of Usage Asssessment

While no dramatic performance improvement was shown in this particular study, the framework for performance improving tests has been created. Enough evidence has been presented that significant throughput can be expected if device drivers or kernel code is written to minimize VU control overhead. The dot-product performance test effectively allowed us to set expectations for the sort of throughput to be expected from the FMAC units in the VPU. This chapter summarizes the basic steps believed to be required to produce satisfactory throughput utilizing the VUs.

If a device driver or more optimized kernel were introduced, then customized code tailored for scientific computing could take full advantage of the vector hardware in the Playstation-2. In short, the primary boundaries preventing exceptionally efficient code are:

- The overhead making a user-mode to kernel-mode transition
- Managing interrupts and threads within the kernel
- Data transfers to and from the VU units

The overhead involved with user-mode to kernel-mode transitions can be eliminated if a software architecture is introduced that allows for coordination of VU utilization entirely within kernel space.

Currently, if a process wants to write information to the VUs using existing Sony Playstation-2 drivers, it needs to memory map the VUs code and data memories and perform a sequence of reads and writes to initialize the VUs, and then issue system calls to instruct the VU to begin operation and signal completion through an interrupt. The overhead of coordinating events is exceptionally high with regards to the amount of work the actual VU is performing. If the software is re-architected in such a manner that event coordination and management is performed in a very streamlined and quick manner entirely within the kernel, then a significant performance improvement could be achieved.

When the VU or VIF signals an interrupt, the Sony device driver searches through a chain of registered event handlers to determine which threads need to be awoken. This is a very general purpose approach, but is entirely inappropriate for extracting optimal performance from the VUs. While the current approach may be appropriate if the VU spends quite a bit of time doing calculations without requiring more control operations, it is not appropriate for tests such as the dot-product performance test.

Finally, the third improvement that would immensely increase overall throughput is if the DMA-to-VIF transfer mechanism were used more effectively. If the information used in scientific calculations were packaged in such a manner that function calls were specified, and blocks of data were inlined with VIF codes, the overall CPU intervention required to manage the VU operation would decrease. This would enable for efficient utilization of the CPU and FPU while simultaneously allowing for intelligent use of the VUs. If this method were combined successfully with the utilization of the graphics synthesizer, it is believed a tremendous throughput would be achieved. It would provide scientific applications with a significant, cost-effective overall performance increase.

Bibliography

Sony Computer Entertainment, Inc. *EE Core Instruction Set Manual*, Sony Computer Entertainment, 2001.

Sony Computer Entertainment, Inc. *EE User's Manual*, Sony Computer Entertainment, 2001.

Sony Computer Entertainment, Inc. *GS User's Manual*, Sony Computer Entertainment, 2001.

Sony Computer Entertainment, Inc. *VU User's Manual*, Sony Computer Entertainment, 2001.

Daniel P. Bovet and Marco Cesati, *Understanding the Linux Kernel.* Sebastopol, CA: O'Reilly and Associates, 2001.

Alessandro Rubini and Jonathan Corbet, *Linux Device Drivers,* Sebastopol, CA: O'Reilly and Associates, 2001.