*Presented at the Thirtieth Annual Asilomar Conference on Signals, Systems, and Computers - November 1996*

# Native Signal Processing on the UltraSparc in the Ptolemy Environment

William Chen, H. John Reekie, Sunil Bhave, and Edward A. Lee

{williamc,johnr,sunil,eal}@eecs.berkeley.edu

Dept. of Electrical Engineering and Computer Sciences, University of California, Berkeley

## Abstract

*We have implemented a number of real-time signal processing kernels and applications within the Ptolemy simulation and code generation environment. Our goal is to make it easy to generate real-time programs with configurable interactive user interfaces. As part of this project, we have developed and benchmarked some key signal processing kernels for the new UltraSparc Visual Instruction Set. We present some performance results and compare the VIS with plain integer and floating-point C code running on the same processor.*

## 1 Motivation

The term "native signal processing" (NSP) has been receiving an increasing degree of attention [4]. The performance of high-performance general-purpose CPUs can exceed that of first- and second-generation digital signal processors (DSPs), and is it now feasible to perform a substantial amount of signal processing on the main CPU of a workstation. Manufacturers have accelerated this trend by including DSP-like instructions on their general-purpose CPUs, including the Intel Pentium MMX [3] and the Sun UltraSparc [10].

We don't expect that embedded DSPs will ever become obsolete: applications with high cost-performance requirements or stringent real-time response requirements will always be more effectively served by dedicated and application-specific processors. Nonetheless, the ability to perform real-time signal processing on the main CPU is increasingly important: the world-wide web and much higher reliance on audio and video capabilities demand much higher performance levels from general-purpose computers.

Sun's new UltraSparc CPU includes a a new set of instructions called the Visual Instruction Set (VIS), specifically geared towards video and image processing [9,10]. In essence, the VIS treats a single 64-bit register as several—two, four, or eight—data words and performs operations on all of these words in a single instruction. Combined with the UltraSparc's ability to issue four instructions per cycle, the architecture allows real-time MPEG decompression to be performed on the workstation's CPU.

Although not specifically designed to support 1-D signal processing, the VIS and UltraSparc appear to be attractive targets for work in this field. The high degree of instruction-level parallelism should allow substantial performance increases on algorithms that can be implemented using fixed-point arithmetic.

We set out to enhance the real-time signal processing capabilities of the Ptolemy enviroment [1], and to evaluate the suitability of the VIS for this purpose. To do so, we incorporated new code generation features into the Ptolemy environment, including a new mechanism for building interactive user interfaces, and added code generation support for the VIS [2].

## 2 The Visual Instruction Set

The goal of the VIS is to speed up the performance of signal processing (mainly image processing) kernels. It has over 50 new instructions and four partitioned data types (figure 1). The data formats treat a 32-bit floating-point register (type "vis_f32") as four eight-bit or two 16-bit words, and a 64-bit floating-point register (type "vis_d64") as eight eight-bit or four 16-bit words. As the data format for all our VIS work, we chose a 4×16-bit word, which we call a "quad-word." This, we felt, was most suited to 1-D work, providing reasonable dynamic range for non-critical audio applications, and minimizing main memory accesses.

Apart from basic arithmetic instructions, the VIS has instructions to pack and unpack data into and out of the partitioned data formats. We will illustrate the instruction set with a parallel 16×16-bit multiply example. The VIS does not have a parallel 16×16-bit multiply instruction, but

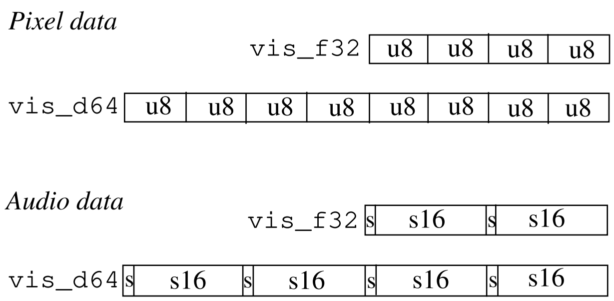


**Figure 1: VIS data formats**

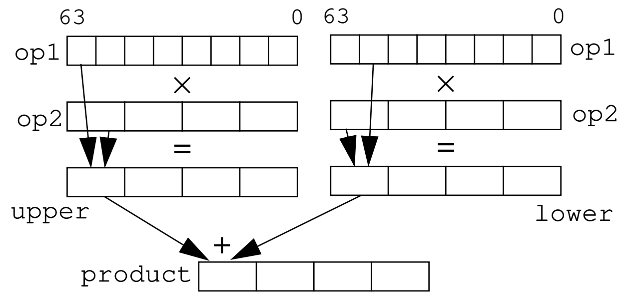


**Figure 2: Parallel VIS 16-bit multiply**

two 8×16-bit multiplies. Figure 2 illustrates how a parallel 16×16-bit multiply is achieved; the code is written:

```
vis_d64 op1, op2, upper;
vis_d64 lower, product;
upper = vis_fpmul8sux16(op1,op2);
lower = vis_fpmul8ulx16(op1,op2);
product = vis_fpadd16(lower,upper);
```

An “instruction” such as *vis_fpadd16* is actually a C function call: the compiler expands these calls to a short sequence of in-line assembler instructions that includes load and store instructions and the actual VIS assembler instruction. Subsequent compiler optimization phases remove redundant loads and stores between VIS instructions.

There is, unfortunately, no instruction in the VIS that can shift each word in a quad-word left by one bit, so fixed-point algorithms that use this instruction sequence must take account of the scaling implied by this operation. (It is also possible to perform two, 16×16-bit multiplies to produce two 32-bit results, in which case scaling is available.)

The VIS includes instructions to align a 64-bit word that is not aligned on a 64-bit address boundary in memory. We do not use these instructions, requiring instead that data arrays be allocated on a 64-bit boundary.

## 3 Implementing Signal Processing Kernels in the VIS

We have found that taking full advantage of the parallel instructions of the VIS requires reformulating each algorithm as a vector-matrix multiplication [6].

We wrote and benchmarked three key kernels for the VIS: the finite-impulse response (FIR) filter, a second-order infinite-impulse-response (IIR) filter, and the Fast Fourier transform (FFT). We will illustrate VIS coding with the FIR filter. The definition of the filter is

$$y[n] = \sum_{k=0}^{n} h[k] \times x[n-k]$$

Each output value from a requires $n$ multiplications and $n$-$1$ additions. By unrolling the convolution and reformulating as a matrix-vector multiplication on quad-words, the filter is as shown in figure 3; potentially, there is a four-times speedup. Because the matrix is essentially a re-ordered (and zero-padded) version of the coefficient array, we create the matrix during initialization of the program. The matrix containing current and past input data ($x$) is allocated on a 64-bit boundary to minimize

$$\begin{bmatrix} y[8] \\ y[9] \\ y[10] \\ y[11] \end{bmatrix} = \begin{bmatrix} h0 & h1 & h2 & h3 & h4 & h5 & h6 & h7 & h8 & h9 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & h0 & h1 & h2 & h3 & h4 & h5 & h6 & h7 & h8 & h9 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & h0 & h1 & h2 & h3 & h4 & h5 & h6 & h7 & h8 & h9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & h0 & h1 & h2 & h3 & h4 & h5 & h6 & h7 & h8 & h9 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x[11] \\ x[10] \\ x[9] \\ x[8] \\ x[7] \\ x[6] \\ x[5] \\ x[4] \\ x[3] \\ x[2] \\ x[1] \\ x[0] \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

**Figure 3: Matrix-vector form of the FIR**

memory accesses.

A portion of the inner loop of the VIS filter code is shown below. This code performs multiplies one row of the matrix by one input quad-word to produce a an output value split into four 16-bit components—to get the value y[8], we will have to sum each of these components:

```
for (outerloop = 0; outerloop < n;
                    outerloop++) {
  data = src[nminusk];
  tapvalue = *tapptr0++;
  /* take inner products */
  pairhi = vis_fmul8sux16(data,
                          tapvalue);
  pairlo = vis_fmul8ulx16(data,
                          tapvalue);
  /* accumulate results */
  pair = vis_fpadd16(pairhi,pairlo);
  accum0 = vis_fpadd16(accum0,pair);
  ...
}
```

In order to produce the output quad-word, the four components of each accumulated value must be split into separate words and summed together. During this process, the data needs to be transferred into integer registers (the cast to type `vis_u32`) so that bit-shifting can be performed. This code is:

```
/* sum accumulators */
splithi = vis_read_hi(accum0);
splitlo = vis_read_lo(accum0);
split = vis_fpadd16s(splithi,
                     splitlo);
accum0u = *((vis_u32*) &split);
splithihi = (short)((accum0u>>16));
splitlolo = (short)(accum0u&0xffff);
y8 = splithihi + splitlolo;
...
```

Finally, we combine the four results into a single quad-word. This requires shifting data from the integer register back to the 64-bit floating-point registers:

```
dlo =(vis_u32)(y9<<16 | y8&0xffff);
dhi =(vis_u32)(y11<<16 | r10&0xffff);
dlofloat = vis_to_float(dlo);
dhifloat = vis_to_float(dhi);
dst1[i] = vis_freg_pair(dhifloat,
                        dlofloat);
```

To measure the performance of the algorithms, we used the UNIX `gethrtime()` high-resolution timer to instrument a loop containing each kernel. Table 1 summarizes the performance we obtained for the VIS, relative to equivalent implementations written in C using both floating-point data and integer data. Speedups over the floating-point implementations range from none for the IIR filter, to over three for the FIR filter.

**TABLE 1.** Performance results

| Kernel | VIS/float | VIS/ integer |
|---|---|---|
| FIR | 3.43 | 6.33 |
| 256-point FFT | 1.28 | N/A |
| Biquad | 0.99 | 2.76 |

Note that the C floating-point algorithms are significantly faster than the C integer algorithms: because the UltraSparc has two integer units and independent floating-point adder, multiplier, and divider, and can issue four instructions per cycle, the floating-point algorithms allow much better utilization of the on-chip resources. Realistic performance comparisons must therefore be performed against the equivalent C floating-point implementation, not against an equivalent integer or fixed-point implementation. For example, some of the performance increases claimed by Sun for the VIS are misleading because they compare with C integer implementations.

The FIR filter exhibits good speedup because of the high parallelism inherent in the matrix-vector formulation of the algorithm, and because the regularity of the algorithm made it relatively easy to partition into parallel four-word instructions. However, because the FIR uses the parallel 16-bit multiply without a left-shift, overall filter has a gain of 0.5. (We also coded a filter with unity gain using the parallel multiply with 32-bit result, but could achieve only a two-times speedup over the floating-point C program.)

The IIR filter exhibits no speedup at all. Despite reformulating the filter into a state-space form [6] to increase the parallelism of the algorithm, and carefully hand-coding the VIS code, the recursive nature of the filter prevents efficient utilization of the parallel 16-bit instructions.

## 4 Real-time signal processing in Ptolemy

Ptolemy supports two key modes of execution: simulation and code generation. In either case, a program is specified as a block diagram—that is, as a dataflow graph. In pure simulation mode, a “star” (block) is coded as a C++ object; at run-time, a pre-generated schedule [5] “fires” stars by calling the *fire()* functions of those objects in a predetermined order. Each star reads from and writes to FIFO buffers.

In code generation mode, hand-coded blocks of code are inserted into a generated program according to a predetermined schedule. Real-time capability for arbitrary

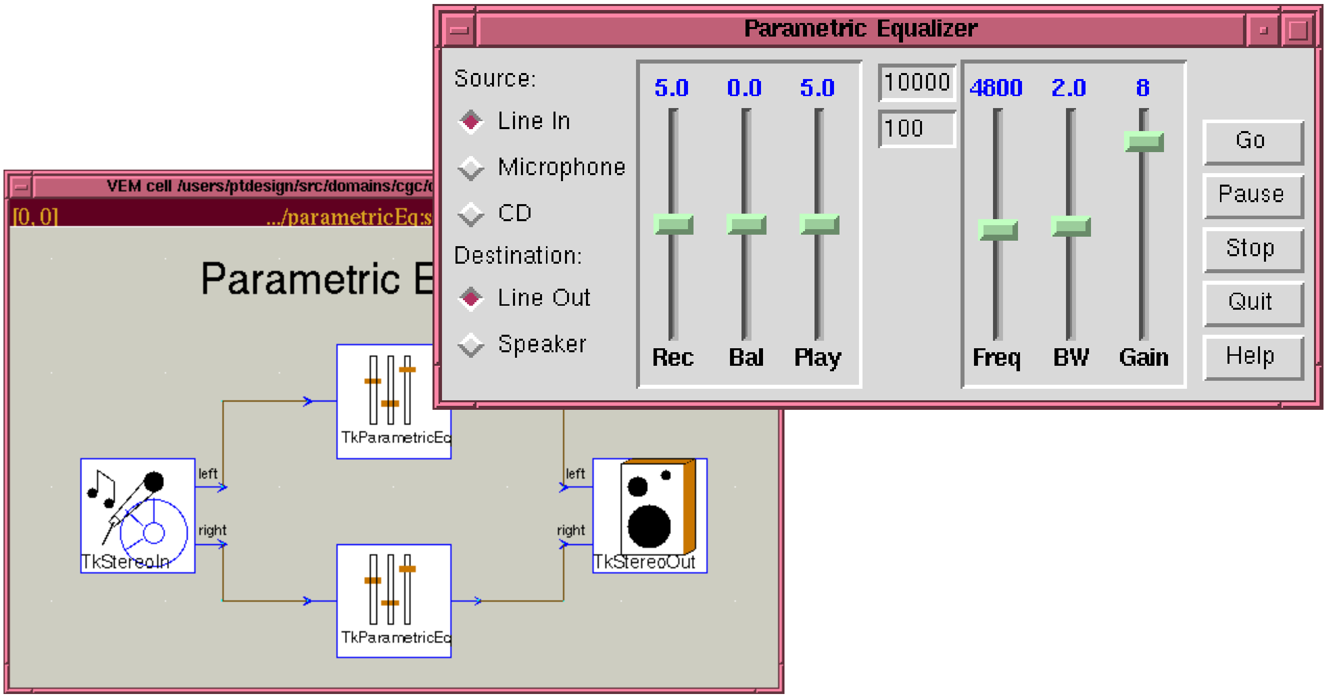


**Figure 4: Parametric equalizer block diagram and user interface**

target architectures is supported—for example, Ptolemy includes a substantial library of code-generation stars for the Motorola DSP56002 processor. We have extended the star library for generation of C code that executes on the host workstation with better support for Sun audio, filtering stars with real-time user control, and stars for VIS code generation.

To provide a basic building-block for audio applications, we implemented a biquadratic filter with closed-form expressions for its coefficients. This enables filter parameters to be updated in real-time without excessive overhead. For the band-pass filters, we used the equations given by Shpak [8]; for the high-pass and low-pass filters, we used a modified version of Shpak's equations (see Chen [2] for the derivation).

Figure 5 shows a simple real-time audio application implemented with these filters: a stereo parametric equalizer. Each filter is a resonant section that can generate a low-pass, band-pass, or high-pass filter. In this example, we have a single band-pass filter on each channel, and provide the user with real-time control over the center frequency, bandwidth (in octaves), and gain (in dB).

To better support real-time user control, we implemented a new interface mechanism in code-generated systems linked in with the Tcl/Tk [7] libraries. Each star and each of its parameters is named; the user can write a Tcl script to build a custom run control panel, either choosing from a library of pre-defined components (such as the banks of sliders and buttons in the example), or coding new ones. Each "control" in the user interface is connected by name to one or more parameters of the stars. At run-time, moving the control changes the star parameters with the appropriate effect.

For example, the example system has stars named *left* and *right*. The slider labeleed "Gain" is connected to the *gain* parameters of the two filter stars, and so on. The user script to create this control panel is about twelve lines of Tcl code.

The example program runs at 44.1kHz with ease on a modern Sun workstation. Using the program "top" on our 140 MHz UltraSparc I gives a CPU utilization of approximately 17%. A more ambitious program, a full ten-band stereo graphic equalizer (figure 5), uses approximately 90% of the CPU on the same processor.

The examples given will run using either the C-coded floating-point filter stars or the VIS versions, with approximately the same CPU utilization. The fixed-point VIS versions do exhibit some instability at high filter Q's, however.

## 5 Conclusions

The performance result for the VIS on the kernels we tested are a little disappointing. Although exhibiting good speedups with the FIR filter, the IIR and FFT kernels exhibited little or none, despite some intensive hand-coding and optimization effort. Nonetheless, for certain applications, there are "free" performance gains available

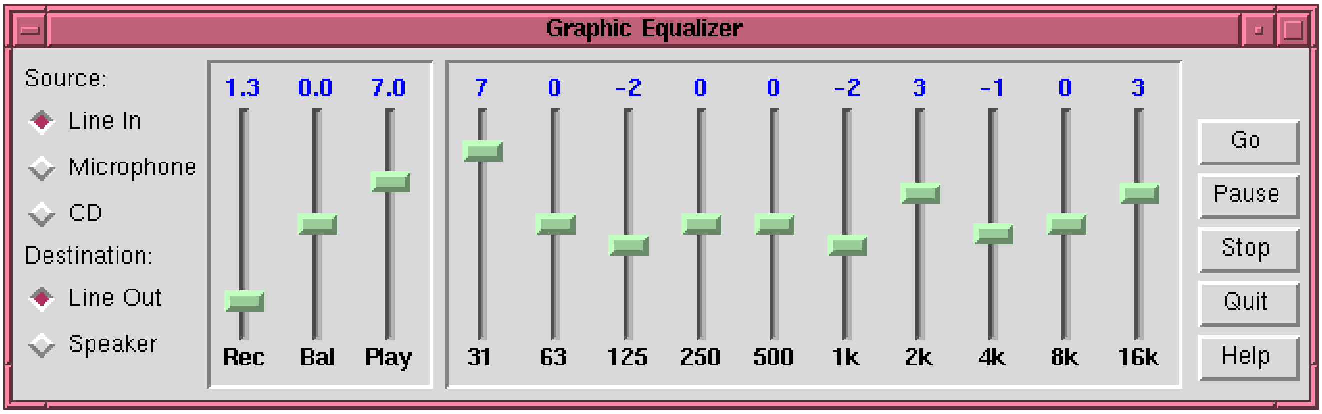


**Figure 5: Ten-band equalizer interface**

on a Ptolemy user's workstation, and we believe we should support these gains where possible.

A drawback of the VIS is the difficulty of coding. Although the function-call syntax means the programmer does not have to manipulate registers directly, the need to manipulate multiple data words in parallel takes some effort on the programmer's part. We venture to suggest that VIS coding is as hard as, if not harder than, typical DSP coding.

We were, however, impressed with the real-time performance we could achieve with the UltraSparc processor—with or without the VIS. Although the ten-band equalizer exhibits audio dropouts as soon as any other work is performed on the workstation, less intensive real-time programs have no such problems. We are currently extending the library of audio stars and user interface components to include a range of common audio processing functions, such as ambience simulation and dynamic range control.

## Acknowledgments

This research is part of the Ptolemy project, which is supported by the Advanced Research Projects Agency and the U.S. Air Force (under the RASSP program, contract F33615-93-C-1317), Semiconductor Research Corporation (project 94-DC-016), National Science Foundation (MIP-9201605), Office of Naval Technology (via Naval Research Laboratories), the State of California MICRO program, and the following companies: Bell Northern Research, Cadence, Dolby, Hitachi, Mentor Graphics, Mitsubishi, NEC, Pacific Bell, Philips, Rockwell, Sony, and Synopsys.

## References

[1] J. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *International Journal of Computer Simulation, special issue on Simulation Software Development*, vol. 4, 1994. http://ptolemy.eecs.berkeley.edu/papers/JEurSim

[2] W. Chen, *Real-time Signal Processing on the Ultrasparc*, Master's Report, Dept. of Electical Engineering and Computer Sciences, University of California, Berkeley, November 1996 (in preparation).

[3] L. Gwennap, "Intel's MMX Speeds Multimedia," *MicroProcessor Report*, vol. 10, issue 3, 1996.

[4] P. Lapsley, "NSP Shows Promise on Pentium, PowerPC," *MicroProcessor Report*, 1995

[5] E.A. Lee and D.G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, 1987.

[6] D. Mitra and J. F. Kaiser, eds. *Handbook for Digital Signal Processing*, John Wiley and Sons., 1993

[7] J.K. Ousterhout, Tcl and the Tk Toolkit, Addison-Wesley 1994.

[8] D.J. Shpak, "Analytical Design of Biquadratic Filter Sections for Parametric Filters," *Journal of the Audio Engineering Society*, vol. 40, no. 11, November 1992.

[9] *UltraSparc-I User's Manual*, Sun Microsystems, 1996.

[10] *Visual Instruction Set User's Guide*, Sun Microsystems, 1995.

[11] D.L. Weaver and T. Germond, eds, *The Sparc Architecture Manual, Version 9*, Prentice-Hall,Inc., 1994.