Virtual Digital Signal Processing in an Object-Oriented System

David K. Mellinger
Center for Computer Research in Music and Acoustics (CCRMA)
Department of Music
Stanford University
Stanford, California USA

davem@Polya.Stanford.edu

G. E. Garnett
Yamaha Music Technologies
Larkspur, California USA

pixar!ymt!guy@ucbvax.Berkeley.edu

Bernard Mont-Reynaud
Center for Computer Research in Music and Acoustics (CCRMA)
Department of Music
Stanford University
Stanford, California USA

davem@Polya.Stanford.edu

Introduction

We describe in this paper a prototype environment for digital signal processing in terms of a *virtual processor* and its associated *virtual data.* The basic operations of the processor are designed for fast vector operations on a variety of different types of hardware. One finished implementation is presented and others are planned.

The modern digital signal processing environment is one of great flux. New chips are being developed and brought to market at ever faster rates. Those creating signal-processing software applications wish to take advantage of the speed of these hardware devices, but are faced with the problem of interfacing each application to each piece of hardware. The work required can be reduced immensely by using a signal-processing software environment that is common to all the low-level hardware devices, but such an environment must have hardware independence "designed in" from the very start.

It is with goal that we have implemented a prototype system for vectorized digital signal processing in the Smalltalk-80 programming system. The development discussed here is the high-level front end to this vector processor. Smalltalk-80 is itself highly machine-independent, due mainly to its being constructed on top of a *virtual machine* (Goldberg and Robson 1983). A primary design choice in the current system has been to abstract the notions of digital signal processors and data, and to provide a standard interface and a virtual machine to support these abstractions. The system is called the Virtual Digital Signal Processing system, or VDSP.

Computer Music Journal, Vol. 13, No. 2, Summer 1989,
© 1989 Massachusetts Institute of Technology.

Virtual Processors and Virtual Data

A *virtual processor* is an object that includes the following functionality: it provides a means for allocating and deallocating data objects; it provides a standard set of vector, scalar, and vector-scalar operations; it provides input/output to external devices capable of handling data formats commonly used in signal processing; and it provides communication between real processors and the user's computing environment.

Virtual data are objects that encapsulate their data and have the following functionality: they provide a handle that is used by the virtual processor to refer to the actual data; they provide representations of useful data types, including floating-point and integer vectors and scalars; and they respond to messages requesting operations on their data by invoking the corresponding operation on a virtual processor.

The example of multiplying two Smalltalk arrays of integers using a virtual processor would be executed using the steps:

1. Send a message telling the virtual processor to allocate two vector objects whose contents are obtained from the two Arrays.
2. Tell one vector object to multiply itself by the other.
3. Tell the result vector to make a new Smalltalk Array whose contents are the data of the result vector.

In real applications, data normally live in virtual objects, so the overhead of transferring arrays of data from and to the user's interactive environment is rarely encountered. The allocation and state of virtual data are shown in Fig. 1.

What VDSP Is Not

The VDSP system does not aim to be an optimizing compiler for the many signal-processing chips available. Rather, it is a portable high-level interface to common signal-processing functionality. The system is not a layered stream-computation system, as is the signal representation language SRL (Kopec 1985). At user request, it performs one operation on one block of data at a time.

Using the Object-Oriented System

Using an object-oriented programming system with class-based inheritance allowed us to develop the VDSP system quickly. One class, *VdProcessor,* contains procedures and data common to all the types of processors available, including methods for transferring data, dispatching arithmetic primitives, error handling, and naming processors. Subclasses of *VdProcessor* represent different types of processors—for example, Motorola 56001 integrated circuits—and instances of these subclasses correspond to actual pieces of hardware. These processor classes have actual primitive calls and also handle memory allocation, which can vary in nature from one type of processor to another. It is possible in the system to be using more than one type of processor at a time. Figure 2 illustrates the class hierarchy of virtual processors.

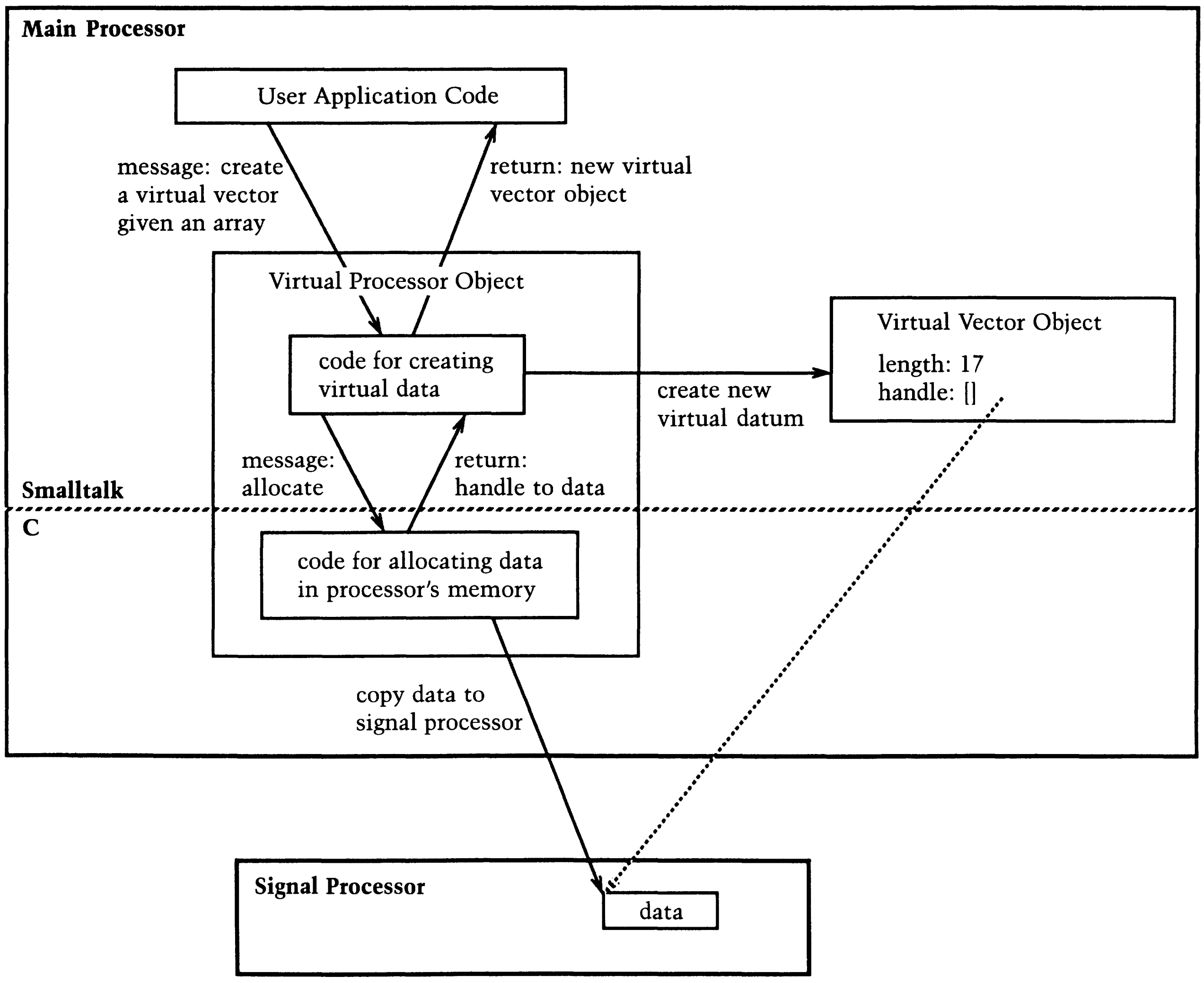
A separate subclass hierarchy holds virtual data objects. Again, a fair amount of common functionality is placed in an abstract class, *VdDatum,* reducing the amount of code duplication needed to implement virtual data objects. This includes type information and coercion, part of the work of instance creation, use of handles, and delegation of arithmetic messages to the appropriate processor object. The subclasses of *VdDatum,* namely *VdVector* and *VdScalar,* have code that is specific to their types, such as size information and vector reduction operations. The class hierarchy of Vd-Datum and an example VdVector instance are shown in Fig. 3.

Another feature of the Smalltalk-80 programming environment was immensely helpful: rapid prototyping. Because of the interactive nature of the environment, we were able to write and test code very quickly, making the time needed for construction of the initial prototype system quite short.

Primitive Operations

Smalltalk currently provides a simple capability to execute external code from within the interactive computing environment. The user writes a piece of

Fig. 1. Allocation and state of a virtual data object.



code in C (or some other low-level language) that will be linked to the Smalltalk virtual machine as a *user primitive.* This primitive can be invoked from within the interactive environment, passing data to and from the C data space and executing the C code of the primitive. This interface mechanism is useful for accessing hardware devices, which are otherwise inaccessible. The virtual processor described here is the high-level, object-based end of such an interface.

There are several types of operation that a virtual processor needs to perform. The primary ones are the vectorized computations that are the *raison d'etre* of the VDSP system. In addition to arithmetic, there are operations for creating (allocating) virtual data objects, for storing and reading objects in external files, and for communicating data objects to and from the user's interactive environment. The set of operations is kept as small as possible for several reasons: a small size makes implementation on a given hardware processor easier and more reliable; it keeps the Smalltalk virtual machine small; and we would rather write code in a high-level object language than in C. On the other

Fig. 2. Class hierarchy for virtual processors showing an instance of a concrete class.

Fig. 3. Class hierarchy for virtual data showing an instance of a concrete class.

Fig. 2

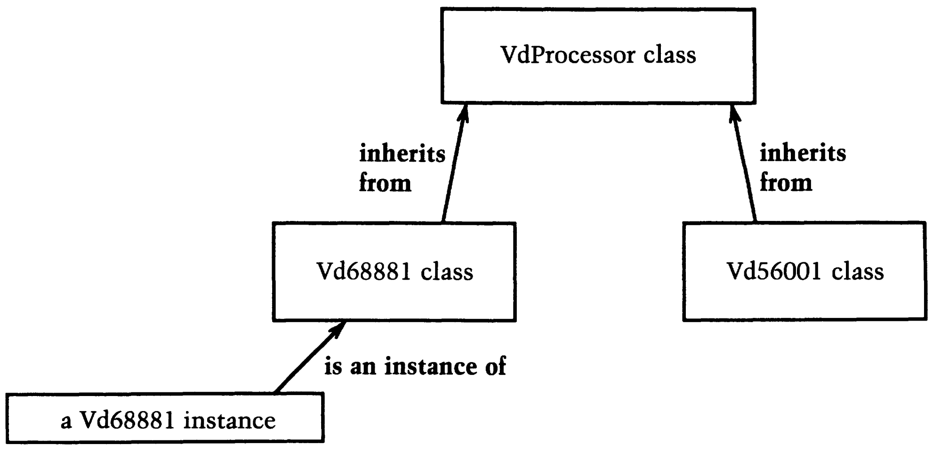
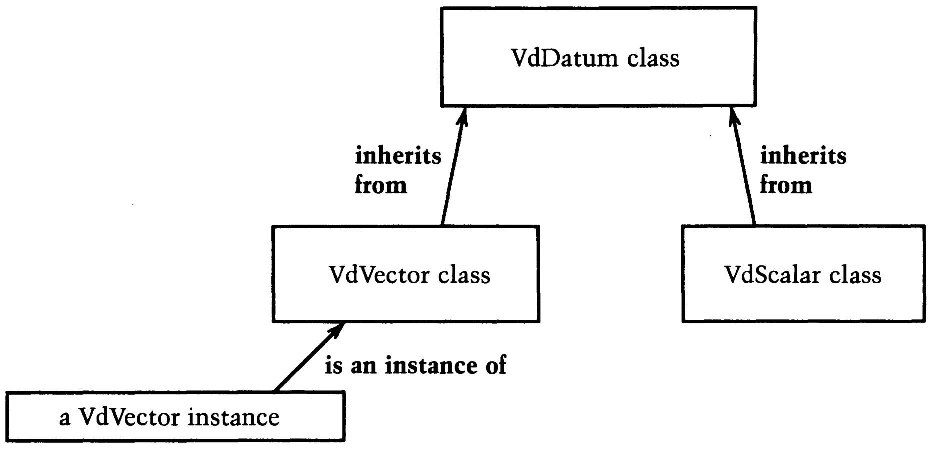


Fig. 3



hand, we want to be able to do all common digital signal processing operations efficiently, so the kernel of operations that are implemented as user primitives must be reasonably complete.

The Primitive Kernel

This kernel is based on arithmetic operations: add, subtract, multiply, divide, negate, reciprocal, and absolute value. Each of these operations is available for vectors and scalars, with the binary operations (+, −, ×, /) working pairwise on the elements of two vectors. Binary operations also have a vector-scalar form so that one can, for example, multiply each element of a vector by a given scalar. The kernel also includes vectorized exponential functions (log, exp, sqrt) and trigonometric functions (sin, cos, arctan). In addition, there are special operations that take a vector and reduce it to a scalar: sum or product of elements, minimum and maximum values, and dot-product. Table 1 summarizes these kernel operations.

Table 1. VDSP kernel primitive operations

Unary arithemtic:	−x, 1/x, |x|, log, exp, sqrt, sin, cos (vector only) min, max
Binary arithmetic:	+, −, ×, /, arctan (vector only) dot-product
Type conversion:	Floor, truncate, asInteger
Memory management:	allocate, free
File I/O:	open, close, read, write, seek, readStream, writeStream

Table 2. Benchmarks for VDSP speed enchancement (operation times, in microseconds)

	Floating point		*Integer*	
Operation	*Smalltalk*	*VDSP*	*Smalltalk*	*VDSP*
+	200	18	120	3
×	200	20	120	6
/	200	22	560	9
Sine	180	36		
Sqrt	1520	16		
Log	3620	40		
Exp	2240	34		

In addition to its normal mode of operation, each arithmetic operation has a "permutation map" version. This version allows one to access elements of a vector by using indirection through a table of indices, making possible arbitrary selection and reordering of vector elements. Some common signal processing functions, such as matrix multiplication and the fast Fourier transform, can be performed quickly with permutation maps.

To support manipulation of virtual data, there are also primitives for allocation and deallocation, and for getting data into and out of virtual data objects. Since vritual data live outside of the Smalltalk environment, their storage space is not automatically reclaimed—as it is for normal Smalltalk objects—and so it must be explicitly allocated and freed by the user. The virtual data objects containing the handles *are* inside the environment, but the data that the handles point to is outside it. Furthermore, virtual data typically do not live through a save/quit/restart cycle in Smalltalk and must be regenerated by the user at the start of each login session.

Finally, there are primitives for storing data to, and reading data from, external files. These are fairly low-level primitives, oriented toward handling various formats of the streams of (audio) samples we are currently working with. There are other primitives for transferring data to and from the user's environment (e.g., turning virtual vectors into Smalltalk Array objects), and for data type conversion.

How Well Does It Work?

The most important goal of the VDSP project is to make a simple, standard interface for signal processing operations so that writing high-level portable code is easy. Though we have not yet had a chance to implement VDSP on more than one hardware platform, we believe that it will be quite easy to carry an application written using VDSP from one implementation to another. Having the abstractions of virtual processor and virtual data has made it easier to write code for vectorized computations in the simple filter and transform applications we have finished to date. The abstractions allow one to separate the operation desired, such as a vectorized multiplication, from the implementation details; such separation greatly helps simplify thinking about the operations.

A second goal is to speed up numerical computation. Smalltalk is not really designed for optimized arithmetic operations the way, say, Fortran is, but VDSP helps get around this problem by allowing for reasonably fast vectorized operations. Table 2 summarizes timings for some common operations in the current implementation. The VDSP timings are shown for the version that uses the Macintosh II's (TM of Apple, Inc.) internal MC68881 math coprocessor for VDSP operations.

The simple VDSP benchmarks measure roughly a factor of 10 improvement for floating-point arithmetic, a factor of 20 or more for integer arithmetic, and varying improvements from 5 to 70 for more complex operations. Note that these improvements are obtained without special hardware. Further speedups will be achieved with the use of dedicated signal processing devices.

A third goal of the VDSP project was to explore the use of object-oriented programming for signal processing. Though our experience is yet slight, it seems as though the object-oriented model can significantly help ease the task of programming. The concepts of virtual processor and virtual data map directly to classes in Smalltalk, so it was easy to write the Smalltalk end of the VDSP implementation. The few applications we have built on top of the system were also simple to implement using the object model and the classes defined by VDSP.

Current Status

The current VDSP implementation runs on an Apple Macintosh II computer using a Smalltalk-80 system from ParcPlace Systems, Inc. for the high-level operations and APDA's MPW C compiler and loader for the low-level primitives. This implementation uses the 68881 coprocessor to do floating-point and integer operations, and the Macintosh II's native operating system for file input/output and memory management. We are also working on another interface for a hardware card from Spectral Innovations, Inc., which contains a special-purpose signal-processing chip, the AT&T DSP-32.

Future Work

The computational model described so far does not include any graphics operations, but applications that require real-time graphics will depend on integrating a graphics kernel with the signal processing kernel, and we have begun work in that direction.

Regarding DSP chips, we are interested in the use of the Motorola 56001. In particular, when CCRMA receives NeXT machines, and Smalltalk becomes available on them, we will be able to put the portability of the VDSP system to another test. This implementation should test the notions of virtual processor and virtual data better than any other, since the 56001 works with different data sizes than either of the implementations finished or under way.

Since the low levels of the VDSP system—the user primitives written in C—operate on objects, it is possible to use object-oriented systems other than Smalltalk for the high level, such as object-oriented extensions of Lisp and C.

We also contemplate building an operation compiler that would speed up the interface slightly and let the user mostly forget about storage allocation and temporary variables. Currently, the user must explicitly allocate space for all temporary vectors used in a computation, which is annoying at best. The operation compiler would let the user specify an *operation block*, a block of Smalltalk code for a complicated vectorized computation. The compiler would preprocess the block, determine what storage needs to be allocated for the block's functions and what VDSP calls need to made to execute it, and store this information. When the user evaluates the block later, the stored information would be used to execute the block operations quickly and with no need for explicit allocation and deallocation of temporaries.

Last, but not least, we plan to bridge the gaps between block-oriented and stream-oriented computation, by developing a "stream of blocks" model, and to extend our model for the use of multiple interconnected signal processors.

References

Goldberg, A., and D. Robson. 1983. *Smalltalk-80: The Language and Its Implementation.* Reading, Massachusetts: Addison-Wesley.

Kopec, G. E. 1985. "The Signal Representation Language SRL." *IEEE Transactions on Acoustics, Speech, and Signal Processing* 33.