# USING JYTHON TO PROTOTYPE AND EXTEND JAVA-BASED SYSTEMS

Dale Parson, Dylan Schwesinger and Thea Steele
Department of Computer Science, Kutztown University
parson@kutztown.edu, {dschw531,tstee423}@live.kutztown.edu

**ABSTRACT**[1]

Jython is an implementation of the interpretive programming language Python written in Java and designed to generate Java Virtual Machine byte codes. It provides easy access to compiled Java classes including substantial Java libraries. It also provides many of the libraries that are popular with programmers of C-based Python. Python's object-oriented language constructs support creation of executable specifications for object-oriented Java systems. Python's high-level source language mechanisms include generic container types, meta-classes, first-class functions, closures, generators, list comprehensions, source-level reflection and run-time interpretation of generated source code. Consequently, Jython can serve as a powerful rapid prototyping and extension environment for Java applications and frameworks. Its integration into the Java Virtual Machine allows construction of layered systems composed of Jython and Java layers. Example systems discussed in the paper include a prototype computer game that uses Java's graphical user interface libraries, a computer music performance system that allows users to write instrument control code as part of a performance, a Jython-Java layered system for analyzing audio data streams, and multiprocessor performance benchmarks written in Jython and Java.

**KEY WORDS**

Extension language, object-oriented software, Python, Java, Jython, rapid prototype.

## 1. Introduction

This report on using the Jython [1,2] implementation of the Python programming language [3,4] to prototype and extend object-oriented Java systems [5] grows out of four projects that the authors have undertaken recently. The first project is a graphical game for a course in advanced object-oriented system design that illustrates the power of using Jython to prototype Java applications while leveraging Java's graphical user interface (GUI) classes and other classes from Java's substantial class library [6]. The second project is a graphical computer music performance application that includes support for *live coding*, an established means of electronic instrument control, that in this application relies on Jython's ability to compile and run extension code at execution time. The third project is a benchmark program that compares performance of a CPU intensive task on two multiprocessor server systems in several multithreaded programming languages including Java and Jython. The fourth project is an investigation into a Java streaming audio library that uses a Java application layer for performance-critical audio analysis tasks and a Jython layer for user interface construction as well as command line exploration of the audio library. Jython, Java and C-based Python are open source languages with freely available implementations of compilers, run-time systems, code libraries, documentation and support communities.

## 2. Related Work

Python, Tcl and Perl are representative *scripting languages* [7] originating in the late 1980's that remain popular. Interpreted languages such as LISP, BASIC and the assorted UNIX Shell languages exerted strong influences on the creation of these language environments. Emphases in using these languages include interpretation and interactive coding, dynamic typing, high-level container types such as associative arrays, and availability of an *eval* function that can interpret generated code at run time. They provide mechanisms for easily invoking and coordinating (a.k.a. "gluing together") compiled executable programs written in *system programming languages* such as C, C++ and Java.

Python and Tcl go further in serving as *extension languages* for application frameworks. While all scripting languages provide means for dispatching and coordinating other programs, an extension language provides software interfaces whereby a developer can load compiled libraries into the extension language interpreter at run time and create application-specific commands for the operations of those plug-in libraries. Scripting languages support program-level granularity in integrating system

[1] Portions of this work were made possible by a PASSHE (PA State System of Higher Education) Faculty Professional Development Grant for the summer of 2010, and by an Instructional Materials Grant from the Kutztown University Professional Development Committee.

programs and other scripts into aggregate programs; extension languages go further in supporting command-level extension of the language itself via an extensible interpreter and dynamic loading of compiled procedures. An extension language caters to the incremental creation of a *domain-specific language* via the loading of domain-specific command libraries into the language interpreter.

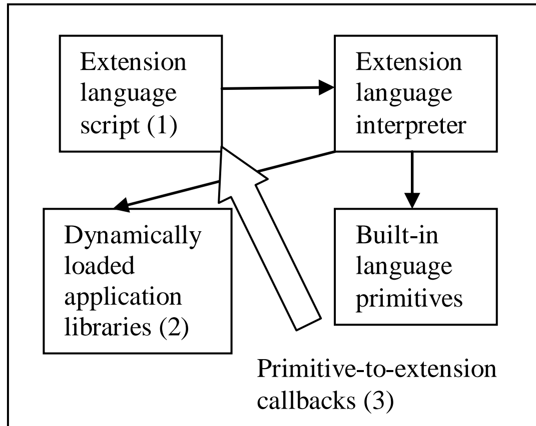


Figure 1: Extension language extension mechanisms

Figure 1 shows three general types of extension supported by extension languages; arrows show direction of procedure invocation. With category 1 the programmer creates extension language procedures that invoke built-in language mechanisms. With category 2 the programmer extends the language's set of built-in commands via dynamic loading of compiled, plug-in extensions. Category 3 consists of callbacks from built-in or plug-in primitives to code written in the extension language. For example, handlers for events triggered in the compiled system programming language can be written in the extension language. An example usage of a category 3 mechanism is the coding of event handlers for graphical user interface (GUI) events in the extension language.

## 3. Jython in Java-based systems

### 3.1 The Jython language and libraries

Jython is an implementation of the Python language specification written in Java. Python is a dynamically typed, interpreted language. The Jython interpreter compiles Python source code into Java byte code at runtime. The Java Virtual Machine (JVM) sees the same result from compilation of either Java or Jython – JVM byte code. However, the code generated from the Jython interpreter will in most cases contain more instructions than a semantically equivalent block of code compiled from Java. This is because dynamically typed languages put off the type decisions until runtime. Statically typed languages handle type checking at compile time, eliminating some instructions that are otherwise necessary to handle type checking at runtime.

The Jython distribution includes the core Python libraries. This inclusion allows for most existing Python code to run on the Jython interpreter with no modification. The real power of Jython over the official C based implementation is the ability not only to leverage existing Python libraries, but also existing Java libraries as well.

The mechanism that achieves this integration of Python and Java code is the *import* statement. Unlike Java's import statement, which is a compiler directive, Jython's import statement is an expression that imports at runtime. Any compiled Java class is available to Jython. During the import process Java reflection accesses class information. Reflection is the examination of objects at runtime to determine all pertinent information, such as data members and methods. As Jython imports Java classes, it converts Java data types into their equivalent Jython types. Writing custom Java code for a Jython framework is far less painful than writing C or C++ code for C-based Python. While the latter activity requires writing low level adapter functions in the system programming language, integrating custom Java code into a Jython framework is accomplished via a simple import statement.

Moreover, the integration infrastructure between Java and Jython also enables utilizing Python modules from within Java code. The most common method to accomplish this is to create a factory method in the Java code that creates an instance of the Jython interpreter, which is itself a Java class, to load the Python module. The module can then be converted from a Python object to a Java object. This integration requires a small amount of effort, but again it is not nearly as troublesome as writing the convoluted wrapper code that is necessary for other extension language implementations in accomplishing similar tasks.

### 3.2 Rapid prototyping for Java-based systems

The key for building a prototype rapidly is the ability to translate high level abstractions into code as effortlessly as possible. The denser the code can be, i.e. the fewer lines of code required to express a concept, the easier it is to create a prototype quickly. A language that does not require much boiler plate code and has high level language abstractions is an ideal candidate for rapid prototyping.

Python is such a language. The syntax of Python is similar to pseudo code, and it is a fairly concise source language. Python is an object-oriented language in the sense that every data type is an object and all of the standard object-oriented abstractions are present, including inheritance and polymorphism. In the case of Jython, these mechanisms offer an easy way to mirror Java class hierarchies. This makes it fairly easy to create a prototype in Jython, and once the high level design decisions are ironed out, the classes can easily be translated into Java equivalents.

Python offers more than just the object-orientated abstractions. It also has implicitly polymorphic built-in complex data types, primarily container objects that can contain heterogeneous data, primitive and aggregate alike. Container types include sequences, associative arrays, and sets. Having these aggregate types built into the language with a compact syntax increases the density of the source code. For example, the following line of Python code sets the variable *a* to refer to a list that contains a string, an integer, and an associative array that maps a string to a string.

```
a = ["Don't panic!", 42, {'key' : 'value'}]
```

While the preceding code could be duplicated in Java, it would require much more effort than the single line of Python code.

Python also supports powerful functional programming features that increase the conciseness of the source language and ability to abstract at a high level. The core feature that supports functional programming is that functions and closures are first-class objects. This means that a function can be passed as an argument to another function, a function can be returned from a function, and a function can be stored in a variable. A closure is a first class function that has one or more free variables bound within its lexical environment. Closures can be used as stateful objects. A full discussion of closures is beyond the scope of this paper. The language also supports lambda expressions and has higher-order functions including *map*, *filter*, and *reduce*, for building and applying composite functions, which are typical higher-order functions in functional programming languages.

Most interpreted languages have the ability to invoke the interpreter during runtime. Python is no exception. A Python program can call the interpreter via *eval* and *exec*. When eval is called on a string that is a valid Python expression, the code is evaluated and the result of the evaluation is returned. The exec statement is similar except any arbitrary string of Python code is executed as if it actually appeared in place of the exec statement. Exec can compile Python class and function definitions for later execution as byte code. Eval and exec allow for powerful metaprogramming capabilities.

### 3.3 Example prototype: successes and pitfalls

In designing a graphical game, we considered a number of possibilities for game play that we could not effectively evaluate without playing the game. Rapid prototyping in Jython let us quickly get to the point of testing the game to make sure that we were on the right track. Had we not used Jython, it is possible that we would have needed to mock up a physical game first. But with Jython, we were able to develop the code and the game play mechanics simultaneously.

Another advantage of using Jython is the fact that the code base is compact and readable, making it easy for someone new to the project to use. Had we asked for outside advice on some aspect of the game play, anyone with the ability to read pseudocode could have understood what was going on. Likewise, there is very little overhead in bringing someone new to the project up to speed.

Many of the benefits above would have been apparent in pure Python as well. One advantage to using Jython is that we were able to add a GUI by leveraging the Java Swing library without rewriting any of the code in Java. In that sense, our prototype was not just a proof-of-concept – we were able to use it to complete a finished looking game. The syntax for using Java objects in Jython is actually more concise than it is in Java. Thus, the GUI code looks a lot cleaner in Jython.

Normally, the trade-off for using a dynamically typed language is a hit to performance due to runtime type checking. In this particular application performance was really not an issue because most of time the game is just waiting for player input. There is another trade-off in the testing phase as well. A statically typed language will catch most type errors at compile time. Programs written in dynamically typed languages need to be more thoroughly tested to ensure that all expressions are providing the correct types. The game is a relatively small amount of code, so in this case the additional testing was also small.

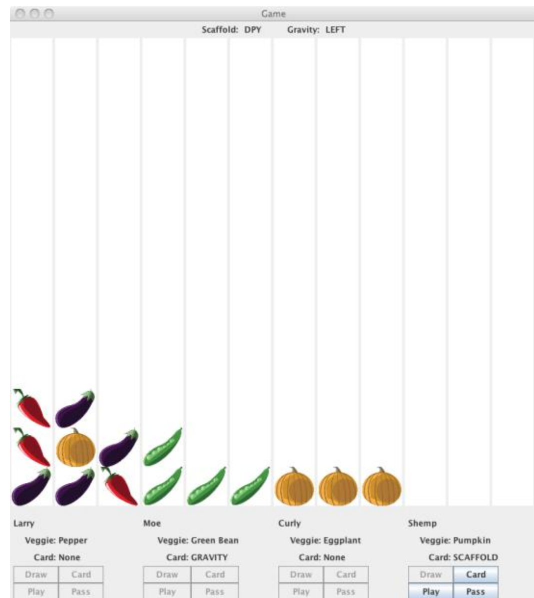


Figure 2: A Java-based graphical, interactive game in Jython

The main pitfall when designing the game was remembering that certain Java types are converted into Jython types. In most cases a given Java object is used like any Jython object, for example using dot notation to call methods. But in the case of strings, the conversion

results in a Python string. So, instead of calling string.length() as is done if the object behaved like a Java object, it must be called as len(string) as in the Python syntax.

### 3.4 Jython as an application-specific language

This section outlines a project that includes support for *live coding*, an established means of electronic musical instrument control, that in this application relies on Jython's ability to compile and run extension code at execution time. The need for this application grew out of a desire to perform synthesized computer music using *just intonation*, an approach to musical scale construction that differs from the twelve-note equal-tempered scale used in most modern Western music [8]. The primary non-graphical mechanisms of this program are construction of tables of per-note frequency information from scale parameters, followed by translation of these frequency table entries into ordered pairs of *note number* and *pitch bend* values for the Musical Instrument Digital Interface (MIDI) protocol [9] used to control music synthesizers. Table construction and translation occur at start-up time using the functional programming mechanisms of Python to convert custom scale parameters to frequency values, and then to convert these frequencies to MIDI values. Java's extensive code library contains packages for the control of MIDI-based synthesizers, including a large array of built-in instrument voice generators. Java's MIDI library can also control software synthesizers from third-party vendors as well as external hardware synthesizers.

Figure 3 is a screenshot of the application's graphical user interface, constructed after data table completion. As with the previous game example, this application uses Java GUI library classes to create the display. It also provides event-handling classes written in Jython that extend Java's event-handling interfaces. These Jython subclasses of Java interfaces react to user actions and periodic timer events.

Each row in the GUI configures performance data for one of 16 MIDI channels in the sound synthesizers. Each channel controls one instrument voice. Graphical controls include buttons that play notes when pressed, spinners that adjust the octave, volume and instrument voice of a row's notes, and combo boxes for several parameters including change of key and target MIDI synthesizer.

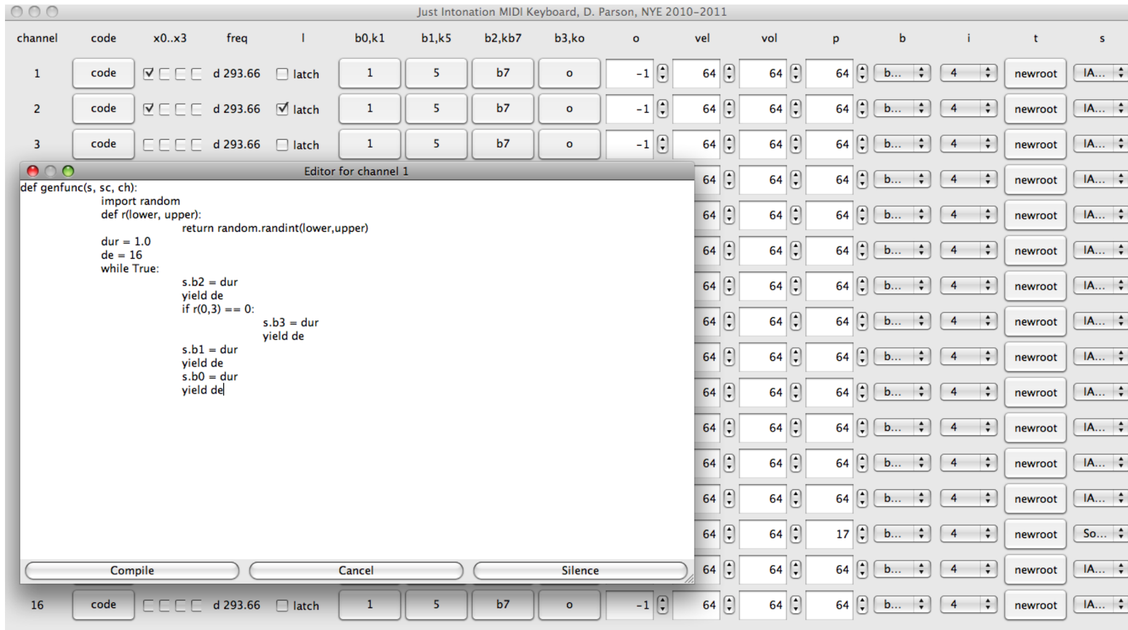


Figure 3: Graphical User Interface for a Jython / Java Just-Intonation Keyboard

MIDI design is based on the Western equal-tempered scale, so that each channel can play only one note at a time in a non-equal-tempered scale, doing so by adjusting the pitch of an equal-tempered note using MIDI *pitch bend* messages. This note-at-a-time constraint, known as *monophonic synthesis*, makes it impossible to play chords and multi-voice parts using standard musical keyboard techniques. Also, a computer mouse supports pressing only one button at a time. Partial solutions to this monophonic constraint include the checkboxes to the left of the note buttons for latching key presses, thereby sustaining notes, along with a 16 x 4 crossbar of checkboxes that allow a button press on a single row to be used in all other rows connected to that row via a crossbar column. The crossbar enables the simultaneous sounding of chords by multiple synthesized instruments, i.e., *polyphony*.

The most important aspect of this musical interface for the current discussion is the use of *live coding*, whereby performers can write and modify snippets of stylized Python code at performance time to control a row of GUI controls in Figure 3 in a way that is reminiscent of a player piano.

```
def genfunc(s): # 'player piano' for MIDI channel
    import random
    def r(lower, upper):
        return random.randint(lower,upper)
    dur = 1.0
    de = 1
    s.p = 80
    s.o = -1
    while True:
        s.b2 = dur
        yield de
        if r(0,3) == 0: # do this 25% of the time
            s.b3 = dur
            yield de
        s.b1 = dur
        yield de
        s.b0 = dur
        yield de
```

Listing 1: Live Python code for controlling the GUI keyboard

Listing 1 shows the code for one MIDI channel that appears in the pop-up code editor window of Figure 3 when a performer clicks that row's "Code" button. This code is pure Python, which the application interprets periodically under the scheduling control of the code itself within the run-time context of the GUI. Listing 1 illustrates that live coding can import any Python library such as the *random* module used here. Mnemonic names that appear across the top row of Figure 3, such as "b2" for note button 2 or "o" for the octave spinner on the GUI, provide the symbols through which a performer controls the GUI via live coding. While a name such as "b2" does not appear to be mnemonic, the use of terse names allows for minimal typing when using the pop-up text editor window during a performance. The appearance of these terse names above their respective columns in the main GUI window makes memorizing them unnecessary.

The code in Listing 1 sets values for local variables *dur* for duration and *de* for delay in temporal units scaled according to a tempo initialization parameter. It sets the channel's instrument voice spinner to MIDI "patch" number 80 via "s.p = 80" and sets the octave spinner using the "s.o = -1" assignment. The "s" symbol represents the scope of the row, and the properties such as ".p" and ".o" represent controls in that row.

After initializing its variables and these two GUI controls, the code of Listing 1 goes into an unbounded loop wherein it presses one of the row's buttons for a duration of *dur* via an assignment statement (e.g., "s.b2 = dur"), then yields control back to the GUI event thread for *de* units of time. The code of Listing 1 comprises a Python *generator*, that acts as a coroutine by yielding control periodically without losing the bindings of its local variables. The program also supports regular Python functions that return the delay until next invocation as a return value at the end of each invocation. The GUI event thread invokes execution of any channel's activated code at its scheduled time under the scheduling control of a Java GUI Timer object. Buttons at the bottom of the editor window of Figure 3 allow a performer to compile and run code after an edit, to cancel an edit or to deactivate the performance code on a channel. The GUI controls on a row remain active for performer interaction even when live code is scheduled for that row.

```
class PJCheckBox(JCheckBox):
    def getValue(self):
        return self.isSelected()
    def setValue(self, value):
        issel = self.isSelected()
        if value and not issel:
            self.doClick()
        elif issel and not value:
            self.doClick()
    v = property(getValue, setValue, None)
```

Listing 2: Java checkbox class adapted in a Jython subclass

Four Python programming constructs make the live coding shown in Figure 3 and Listing 1 possible. First is the creation of Jython subclasses for the Java GUI control classes that add the ability to read and write their values as simple integers and strings via getValue and setValue methods. Listing 2 shows an example subclass of Java's JCheckBox class. These subclasses allow the GUI controls to provide an identical pair of getValue / setValue methods, regardless of the GUI control class. Subclasses serve as adapters that make reading and writing controls homogeneous with respect to method names, number of parameters and return values. Python's dynamic typing makes this task easier to achieve than it is with Java, because the methods parameter types of Java GUI controls vary according to control class, although it could be done via java.lang.Object parameter and return types and explicit type checking in Java.

Jython's subclass of Java's JButton control class was the most complicated control class to write because of the need to emulate timed button presses and releases across rows tied together with a crossbar, but after completion the button presses could be emulated in software or triggered by user actions.

The line "v = property(getValue, setValue, None)" at the bottom of Listing 2 shows the second of the four pertinent Python constructs, which is the ability to create *class properties* whose getValue / setValue methods are invoked when the property is read / written respectively within a Python expression. In the object representing a row in Figure 3, all of the GUI controls are accessible via mnemonic property names appearing across the top row, such as "s.b2" which is the b2 property of the row, this

property being tied to getValue / setValue method pairs for the button object labelled “b2.” Tying GUI control objects to mnemonic property names makes it possible for GUI values to appear as *rvalues* (implicitly invoking getValue) in expressions and as *lvalues* (implicitly invoking setValue) in assignment statements such as those appearing in Listing 1. The result is minimalist syntax for a musician transforming live code during performance, avoiding the more complex, alternative function invocation syntax.

The third pertinent Python construct is support for adding new fields, methods and properties to individual Python objects as well as their defining classes at run time. This program has a class called *environ* with sixteen instances, one per MIDI channel. The “s” object reference manipulated by statements such as “s.b2 = dur” in Listing 1 is an environ object reference. The environ class provides a method for adding properties such as PJButton “b2” at run time as the GUI is constructed. This mechanism simplifies tying together GUI object construction with extension code access to GUI objects via incremental addition of these objects as symbolic properties to the environ objects that are manipulated by the live code.

The fourth and most powerful construct is application of Python’s *exec* function for compiling live coding source functions, such as *genfunc* of Listing 1, at execution time within the scope of class *environ* and the MIDI channel’s scope object. A musician’s live code is compiled to byte code via *exec* and then placed on a scheduling queue that is maintained by Java’s GUI Timer class. Compilation and scheduling occur when a performer clicks the Compile button at the bottom of the code editor window of Figure 3. In this way user scripts can access GUI controls as symbolic names within algebraic expressions as they appear in Listing 1 (s.x0, s.b0, etc.), and the underlying machinery outlined above takes care of the work of converting appearance of these symbols-in-expressions into method invocations.

In a Java program without extension language support it would be necessary to design a custom live coding language, capture its grammar, build a scanner and parser using custom code or a complex parser generator package, compile live code functions, then schedule and interpret their execution. Thanks to Jython’s support for run-time compilation and execution of Python source code, this project has avoided the creation of a custom language, along with the compiler and run-time support for its execution. This live coding facility grows directly out of Python, extending it into being a domain-specific language for live coding.

### 3.5 Layered systems and performance

This section looks at performance issues from two perspectives. The first is processing speed on two shared-memory multiprocessors running multithreaded Jython, Java and C++ code. The second is an overview of a two-tiered approach to using an extension language, where Jython serves to configure and direct time-critical signal processing activities occurring in a Java thread.

The first program to discuss is one of a series of benchmarks exploring the performance of several multithreaded algorithms using several programming languages on two shared-memory multiprocessor servers, a 64-processor Sun Sparc server with limited per-core cache [10] and a 16-processor Advanced Micro Devices server with copious per-core cache [11]. Table 1 shows execution results in seconds for running a multithreaded solution to the N Queens Problem that finds all solutions on a 15 x 15 chess board using Jython, Java and C++ on these two multiprocessors. N Queens represents a search problem with a high number of search states, but with limited demands on memory. The algorithm spawns parallel search threads when advancing to adjacent columns in the board, with the number of threads being a function of number of concurrent columns, configured as a run-time parameter. Table 1 shows execution seconds as a function of the number of software threads for each machine-programming language pair.

| **threads** | **1** | **15** | **182** | **1764** |
|---|---|---|---|---|
| Sparc C++ | 100.54 | 8.07 | 4.37 | 5.87 |
| Sparc Java | 112.602 | 8.936 | 6.253 | 5.391 |
| Sparc Python | 19028.483 | 1517.602 | 745.118 | 737.132 |
| Sparc Jython | 24001.803 | 9654.617 | 12164.258 | 15132.37 |
| AMD C++ | 16.12 | 5.71 | 3.59 | 3.82 |
| AMD Java | 29.867 | 2.916 | 2.316 | 3.446 |
| AMD Python | 2825.602 | 211.302 | 179.349 | 178.108 |
| AMD Jython | 5807.911 | 11283.732 | 7065.99 | 9025.306 |

Table 1: Search time seconds as a function of thread count

The most obvious result is that Jython is the slowest of the languages on both hardware platforms, despite the fact that Jython generates and executes Java byte code. Jython is slower even than C-based Python. This fact is interesting because C-based Python does not support concurrent execution of multiple threads within the Python interpreter. It is necessary to spawn multiple Python processes in order to achieve concurrent use of a multiprocessor. Jython, which supports multithreading within a single process, is nevertheless much slower. Also, its multithreading performance is worse than its single threaded performance on the AMD machine.

Perhaps a more important result is the fact that Java execution speed is on par with C++ execution speed on both platforms, in fact beating C++ performance in

multithreaded AMD runs. This result appears in other benchmarks as well. In many cases multithreaded Java surpasses optimized, multithreaded C++ in speed on the Sparc machine. This result appears to be attributed largely to advances in run-time, profiling-based optimization by the Java just-in-time compiler that translates Java byte codes to machine code during execution.

The conclusion, that Java execution has become fast enough to deal with some classes of real-time processing including audio signal processing, led to the architecture of the final project of this study. Diagrammed in Figure 4, this two-tiered software architecture uses Jython to configure a signal processing thread, coded in Java, that reads input audio streams, transforms them, for example by mixing, adding delays and other effects, and sends them to output streams. Jython's first use in this architecture has been as a means to explore the javax.sound.sampled Java library interactively. This library includes reflective data access methods that describe the audio input and output streams in symbolic terms. A programmer can use Jython interaction with reflective audio classes, in conjunction with the library documentation, to learn how to use library components. After that the programmer can write Jython GUI and configuration code that interacts with Java components in configuring a Java signal processing thread by interconnecting Java signal processing objects used by the Java thread. Jython can access locks and other Java objects from java.lang.concurrent in synchronizing construction of an audio signal processing flow. This two-tiered architecture is an ideal match for the relative strengths and weaknesses of Jython and Java. Jython provides interactive access to reflective (self-describing) classes and fast coding for constructing a GUI and configuration classes, and Java provides fast execution of a signal processing thread.

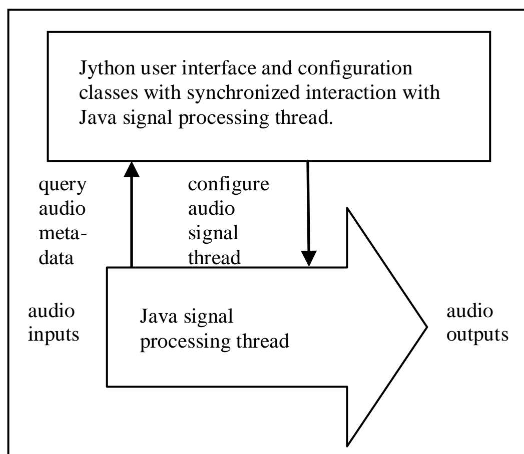


Figure 4: A two-tiered Jython-Java audio signal processor

## 4. Conclusion

Working with Jython and exploring its abilities to create Java-based systems rapidly has been exciting. Standard Python libraries are available in Jython, and any compiled Java class including the massive Java standard library is available via the use of a simple Python import statement. Jython supports use of Java classes, and it supports subclass extension of Java interfaces and classes in the form of Jython classes. It is even possible to supply Jython classes as event handlers for Java-triggered events without writing any Java code.

Python as a source language supplies object-oriented constructs that make it a good fit for specifying and prototyping object-oriented Java systems. Python also supplies powerful functional programming constructs and generic container types built into the language. These capabilities make Python a powerful vehicle for rapid prototyping by enabling concise, specification-like construction of prototype code.

Jython's support for interactive, run-time compilation of Python source code procedures into Java byte code means that Jython can serve as a domain-specific language for an application written in Java. Interpretation, compilation and extension mechanisms built into Python and Jython eliminate any need to create custom domain-specific languages, and to design and build special tools for their support.

Finally, the advent of just-in-time Java compilers that generate code that performs on par with optimized C++ code means that Java has become a viable language for writing real-time systems. In such a system Jython serves the needs of user interface construction and system configuration, applying the interactive strengths of Jython in working with users while leaving performance critical "heavy lifting" to Java.

## References:

[1] J. Juneau, J. Baker, F. Wierzbicki, L. S. Muñoz, & V. Ng, *The definitive guide to Jython: Python for the Java platform* (New York, NY: Apress, 2010). An open source version is at http://www.jython.org/jythonbook/en/1.0/.

[2] The Jython Project, http://www.jython.org/, February, 2011.

[3] D. Beazley, *Python essential reference*, fourth edition (Reading, MA: Addison-Wesley, 2009).

[4] Python Programming Language – Official Website, http://www.python.org/, February, 2011.

[5] K. Arnold, J. Gosling, & D. Holmes, *Java™ programming language*, fourth edition (Upper Saddle River, NJ: Prentice Hall, 2005).

[6] Oracle Corporation, *Java™ Platform, Standard Edition 6 API Specification*, http://download.oracle.com/javase/6/docs/api/index.html, February, 2011.

[7] J. Ousterhout, Scripting: Higher-Level Programming for the 21st Century, *IEEE Computer 31*(3), 1998, 22-30.

[8] G. Loy, *Musimathics, the mathematical foundations of music* (Cambridge, MA: MIT Press, 2006).

[9] MIDI Technical Fanatic's Brainwashing Center, February 2011, http://www.blitter.com/~russtopia/MIDI/~jglatt/.

[10] Fujitsu, Sparc Enterprise T5120, T5220, T5140 and T5240 Server Architecture, http://www.fujitsu.com/downloads/SPARCE/whitepapers/T5x20-T5x40-wp-e-200907.pdf, July 2009, URL verified February, 2011.

[11] Sun Microsystems, Sun Fire™ T1000 and T2000 Server Architecture, http://www.sun.com/servers/x64/x4600/arch-wp.pdf, December, 2005, URL verified February, 2011.